1 Transport

AMQP defines a peer-to-peer protocol for transferring messages between nodes in the AMQP network. This portion of the specification is not concerned with the internal workings of any sort of node, and only deals with the mechanics of unambiguously transferring a message from one node to another.

The AMQP peer-to-peer protocol operates over any underlying network protocol that provides an ordered stream of bytes. The specification defines a grammar used to parse the byte stream into distinct frames. These frames then carry the commands and controls exchanged by the two peers.

The AMQP Network consists of Nodes connected via Links. Messages can originate, terminate, or be relayed by nodes, and the Link protocol manages the transfer of responsibility as messages pass between two Nodes. Nodes exist within a container, and each container may hold many nodes. Examples of AMQP Nodes are Producers, Consumers, and Queues. Producers and Consumers are the elements within a client Application that generate and process messages. Queues are entities within a broker that store and forward messages. Examples of containers are brokers and client applications.

Nodes are expected to have well known names whose format and precise semantics are intentionally opaque to the transport. Care is given to permit the use of long names with no significant loss of efficiency, therefore it is reasonable for node names to be both human readable and globally unique.

A Link transports Messages from one Node to another. Links may be established with an optional filter that controls which messages are permitted to pass through the Link. Where the two nodes are on different peers, the Link protocol is used to transfer the message. The Link protocol is defined as a set of Commands carried over a Session. Multiple Links may be established over a single Session.

A Session is an ordered peer to peer Command transport that optionally retains conversational state when disconnected. AMQP peers interact via a Session. Sessions are expected to map to a single thread of control. Sessions provide a scope for transactional interactions, and a facility for demarcating transactional units of work.

A Command is a discrete payload that is reliably transported by the Session. The local Application will pass a Command to the local Session endpoint. This Command will be transmitted to the remote Session endpoint. The remote endpoint will subsequently dispatch the Command to the remote Application for execution. Once the Command is executed, the remote Application will either acknowledge the success of the Command or report any errors to the remote Session endpoint. The two endpoints then exchange execution state as necessary.

Controls carry information between the two communication endpoints. Unlike Commands, controls are not dispatched to the Application using the Session, but instead are handled by the Session endpoint directly. Controls are used to negotiate Connection parameters, authenticate the Connection, and to establish the transient relationship between a Session and its Connection.

Connections carry frames for multiple Sessions. A frame is an encoded Command or Control. A Connection endpoint may have zero or more associated Sessions. Both Connection endpoints assign each associated Session an outgoing channel number that serves as an alias for the Session in all outgoing frames sent by that endpoint. Each Connection endpoint maintains a map from incoming channel number to Session. This map is used to demux incoming frames and direct them to the correct Session. The mapping between the incoming channel number and Session is established when the `attach` control is received by the Connection endpoint. The mapping between the outgoing channel number and the Session is chosen freely by the Connection endpoint. Note that because each Connection endpoint independently chooses its alias, incoming and outgoing frames for a given session may use different channel numbers.

A key difference between a Command transport and a datagram transport is that unlike data delivery, Command execution can fail. The Session protocol is designed to handle this possibility.

Command execution failure is handled as similarly as possible to network failure. This ensures there is only one recovery procedure for many distinct failure modes. When command execution fails, the Session is detached just as if a network failure had occurred. The key difference is that when a Command execution fails, an explicit detach Control is sent. The detach Control communicates precise information about which Command failed and why. The Sender of the Command can then choose to terminate the Session or recover from the last successfully executed Command as if from a network failure.

- Messages are proxied through the Session via the transfer Command. Likewise Message acknowledgements are proxied as acknowledgements to the transfer Command.
- Each link can be configured with independent flow control of message transfers, or if desired the link flow control can be disabled. Links with flow control disabled are still subject to flow control of Commands by the Session.

```
+------------+
|    Link    |  Message Transport (Node to Node)
+------------+
     /|\ 0..*
      |
      |
      |
     \|/ 0..1
+------------+
|  Session   |  Command Transport (Host to Host)
+------------+
     /|\ 0..*
      |
      |
      |
     \|/ 0..1
+------------+
| Connection |  Frame Transport (Host to Host)
+------------+
```

2 Framing

2.1 Frame Layout

A frame encodes a control or a command. Each frame is divided into two distinct areas: a fixed width frame header, and a variable width frame body containing, when non-empty, an encoded command or control operation.

```
+--------------+------------+
| frame header | frame body |
+--------------+------------+
    24 bytes     *variable*
```

frame header: The frame header is a fixed size (24 byte) structure that precedes each frame. The frame header includes information required to parse the rest of the frame, as well as key session state variables.

frame body: The frame body, when non-empty, contains a command or control operation encoded as a struct.

2.2 Frame Header

```
        +0         +1         +2         +3
   +-------------------------------------------+
 0 |                   size                    |
   +-------------------------------------------+
 4 |  type  |   flags    |       channel       |
   +-------------------------------------------+ -.
 8 |               acknowledged                |  |
   +-------------------------------------------+  |
12 |                 executed                  |  |
   +-------------------------------------------+  |
16 |                 capacity                  |  |
   +-------------------------------------------+  |---+
20 |                command-id                 |  |   |
   +-------------------------------------------+ -'   |
                                                      |
            type: 0x00 - control                      |
                  0x01 - command                      |
                                                      |
           flags: 0|0|0|0|0|0|0|0 - control           |
                  0|0|0|0|0|0|S|F - command           |
                                                      |
               S: sync flag                           |
               F: flush flag                          |
                                     -.               |
              reserved      - control |---------------+
              session state - command |
                                     -'
```

size: Bytes 0-3 of the frame header contain the frame size. This is an unsigned 32-bit integer that MUST contain the total frame size including the frame header. The frame is malformed if the size is less than the the size of the header (24 bytes).

type: Byte 4 of the frame header is a type code. The type code indicates the format and purpose of the frame. A type code of 0x00 indicates that the frame is a control frame. A type code of 0x01 indicates that the frame is a command frame. The subsequent bytes in the frame header may be interpreted differently depending on the type of the frame.

flags: Byte 5 of the frame header is reserved for frame flags. For controls, all bits are reserved. For commands, bits 2-7 are reserved, bit 0 is defined as the flush flag, and bit 1 is defined as the sync flag. All bit numbers are start at 0 for the least significant bit, and increase to 7 for the most significant bit.

sync flag: The sync flag is a signal to the session endpoint to promptly send the newly updated session state after the framed command has been executed.

flush flag: The flush flag is a signal to the session endpoint to promptly send the current session state.

channel: Bytes 6 and 7 of the frame header contain the channel number. The channel number uniquely identifies one of the sessions associated with the connection. An implementation MAY service incoming frames on distinct channels in any desired order. Each peer SHOULD balance the traffic on all active channels in a fair fashion.

acknowledged: Bytes 8-11 of a command frame contain the acknowledged field. This field indicates which outgoing commands have been acknowledged. The acknowledged field is part of the session state as defined in the session section. For controls this field is unused and marked as reserved.

executed: Bytes 12-15 of a command frame contain the executed field. This field indicates which incoming commands have been executed. The executed field is part of the session state as defined in the session section. For controls this field is unused and marked as reserved.

capacity: Bytes 16-19 of a command frame contain the capacity field. This field indicates how many more incoming commands are permitted beyond those that have already been executed. This capacity field is part of the session state as defined in the session section. For controls this field is unused and marked as reserved.

command-id: Bytes 20-23 of a command frame contain the command-id field. This field contains the id of the next command to be sent. If the frame contains a command this is the id of the framed command. If the frame does not contain a command (the frame has no body), then this contains the id that will be assigned to the next command sent. See the session section for more details. For controls this field is unused and marked as reserved.

2.3 Frame Body

The frame body encodes a command or control operation. The operation is encoded as a struct that identifies both the command or control being sent, as well as the field values for the operation. The encoding of a struct is fully defined in the basic-types section.

```
   +-------------------------------------------+
24 |                 op-size                   |
   +-------------------------------------------+
28 |      op-code         | field-packing-flags |
   +-------------------------------------------+
32 |                  fields                   |
   |                    :                      |

   |                    :                      |
   +-------------------------------------------+
```

2.4 Empty Frames

Empty control frames are used to generate artificial traffic as needed to satisfy the negotiated heartbeat interval.

Empty command frames are used to exchange session state variables without sending an additional command. The sync bit when set on an empty command frame is semantically equivalent to retroactively setting the sync flag on the preceeding command.

3 Connection

3.1 Version Negotiation

Prior to sending any frames on a connection, each peer MUST start by sending a protocol header that indicates the protocol version used on the connection. The protocol header consists of the upper case ASCII letters "AMQP" followed by 1, 0, 0, and connection flags. This is an 8-octet sequence:

```
  4 OCTETS   1 OCTET   1 OCTET   1 OCTET    1 OCTET
+----------+---------+---------+----------+---------+
|  "AMQP"  |  major  |  minor  | revision |  flags  |
+----------+---------+---------+----------+---------+

    flags: 0|0|0|0|0|0|R|S

        R: tls-required  -- The peer requires all data following the
                            connection header to be encrypted. If R is
                            set then S MUST also be set.

        S: tls-supported -- Indicates that the peer supports TLS
                            encryption. If either peer's connection
                            header has the R flag set then all
                            connection data following the connection
                            header will be encrypted, otherwise
                            connection data will be unencrypted.

        Note: if one protocol header requires TLS, and the other does
            not support it, then both peers MUST abort the connection.
```

An AMQP client and server agree on a protocol version as follows:

- When the client opens a new socket connection to an AMQP server, it MUST send a protocol header with the client's preferred protocol version.
- If the requested protocol version is supported, the server MUST send its own protocol header with the requested version to the socket, and then implement the protocol accordingly.
- If the requested protocol version is **not** supported, the server MUST send a protocol header with a **supported** protocol version and then close the socket.
- When choosing a protocol version to respond with, the server SHOULD choose the highest supported version that is less than or equal to the requested version. If no such version exists, the server SHOULD respond with the highest supported version.
- If the server can't parse the protocol header, the server MUST send a valid protocol header with a supported protocol version and then close the socket.

Based on this behavior a client can discover which protocol versions a server supports by attempting to connect with its highest supported version and reconnecting with a version less than or equal to the version received back from the server.

Version Negotiation Examples

```
TCP Client                                 TCP Server
======================================================
AMQP%d1.0.0.0        ------------->
                     <-------------       AMQP%d1.0.0.0 (1)
                           ...            *proceed*

AMQP%d1.1.0.0        ------------->
                     <-------------       AMQP%d1.0.0.0 (2)
                                          *TCP CLOSE*

HTTP                 ------------->
                     <-------------       AMQP%d1.0.0.0 (3)
                                          *TCP CLOSE*
------------------------------------------------------
  (1) Server accepts connection for: AMQP, major=1,
      minor=0, revision=0, flags=0

  (2) Server rejects connection for: AMQP, major=1,
      minor=1, revision=0, flags=0. Server responds
      that it supports: AMQP, major=1, minor=0,
      revision=0, flags=0

  (3) Server rejects connection for: HTTP. Server
      responds it supports: AMQP, major=1, minor=0,
      revision=0, flags=0
```

Please note that the above examples use the literal notation defined in RFC 2234 for non alphanumeric values.

Definition: PORT

Value: *5672* ***Description:*** *the IANA assigned port number for AMQP*

The standard AMQP port number that has been assigned by IANA for TCP, UDP, and SCTP.

There is currently no UDP mapping defined for AMQP. The UDP port number is reserved for future transport mappings.

Definition: MAJOR

Value: *1* ***Description:*** *major protocol version*

Definition: MINOR

Value: *0* ***Description:*** *minor protocol version*

Definition: REVISION

Value: *0* ***Description:*** *protocol revision*

3.2 Opening a Connection

Each AMQP connection begins with an exchange of capabilities and limitations. After establishing or accepting a TCP connection and sending the protocol header, each peer must send an `open` control before sending any other frames. The `open` control describes the capabilities and limits of that peer. After sending the `open` control each peer must read its partner's `open` control and must operate within mutually acceptable limitations from this point forward.

```
TCP Client              TCP Server
==================================
TCP-CONNECT             TCP-ACCEPT
PROTO-HDR               PROTO-HDR
OPEN        ---+   +--- OPEN
                \ /
wait             x      wait
                / \
proceed     <--+   +--> proceed

                ...
```

3.3 Pipelined Open

For applications that use many short-lived connections, it may be desirable to pipeline the connection negotiation process. A peer may do this by starting to send commands or controls before receiving the partner's connection header or `open` control. This is permitted so long as the pipelined commands and controls are known a priori to conform to the capabilities and limitations of its partner. For example, this may be accomplished by keeping the use of the connection within the capabilities and limits expected of all AMQP implementations as defined by the specification of the `open` control.

```
TCP Client                    TCP Server
===============================================
TCP-CONNECT                   TCP-ACCEPT
PROTO-HDR                     PROTO-HDR
OPEN              ---+   +--- OPEN
                      \ /
pipelined cmd/ctl      x      pipelined cmd/ctl *
                      / \
proceed           <--+   +--> proceed

                      ...
-----------------------------------------------

  * Note that a peer's use of pipelined
    commands and/or controls cannot be
    observed by the partner so long as the
    pipelined commands and controls conform
    to the partner's capabilities and
    limitations.
```

3.4 Closing a Connection

Prior to closing a connection, each peer must write a `close` control with a code indicating the reason for closing. This control must be the last thing ever written onto a connection. After writing this control the peer should continue to read from the connection until it receives the partner's `close` control.

```
TCP Client         TCP Server
=============================
            ...

CLOSE     ------->
               +-- CLOSE
              /    TCP-CLOSE
TCP-CLOSE <--+
```

3.5 Simultaneous Close

Normally one peer will initiate the connection close, and the partner will send its close in response. However, because both endpoints may simultaneously choose to close the connection for independent reasons, it is possible for a simultaneous close to occur. In this case, the only potentially observable difference from the perspective of each endpoint is the code indicating the reason for the close.

```
TCP Client           TCP Server
================================
              ...

CLOSE     ---+   +--- CLOSE
              \ /
               x
              / \
TCP-CLOSE <--+   +--> TCP-CLOSE
```

3.6 Connection States

START: In this state a connection exists, but nothing has been sent or received. This is the state an implementation would be in immediately after performing a socket connect or socket accept.

HDR_RCVD: In this state the connection header has been received from our peer, but we have not yet sent anything.

HDR_SENT: In this state the connection header has been sent to our peer, but we have not yet received anything.

OPEN_PIPE: In this state we have sent both the connection header and the `open` control, but we have not yet received anything.

OC_PIPE: In this state we have sent the connection header, the `open` control, any pipelined connection traffic, and the `close` control, but we have not yet received anything.

OPEN_RCVD: In this state we have sent and received the connection header, and received an `open` control from our peer, but have not yet sent an `open` control.

OPEN_SENT: In this state we have sent and received the connection header, and sent an `open` control to our peer, but have not yet received an `open` control.

CLOSE_PIPE: In this state we have send and received the connection header, sent an `open` control, any pipelined connection traffic, and the `close` control, but we have not yet received an `open` control.

OPENED: In this state the the connection header and the `open` control have both been sent and received.

CLOSE_RCVD: In this state we have received a `close` control indicating that our partner has initiated a close. This means we will never have to read anything more from this connection, however we can continue to write commands/controls onto the connection. If desired, an implementation could do a TCP half-close at this point to shutdown the read side of the connection.

CLOSE_SENT: In this state we have sent a `close` control to our partner. It is illegal to write anything more onto the connection, however there may still be incoming controls and/or commands. If desired, an implementation could do a TCP half-close at this point to shutdown the write side of the connection.

END: In this state it is illegal for either endpoint to write anything more onto the connection. The connection may be safely closed and discarded.

3.7 Connection State Diagram

The graph below depicts a complete state diagram for each endpoint. The boxes represent states, and the arrows represent state transitions. Each arrow is labeled with the action that triggers that particular transition.

```
             R:HDR @=======@ S:HDR             R:HDR[!=S:HDR]
          +--------| START |-----+    +--------------------------------+
          |        @=======@     |    |                                |
         \|/                    \|/   |                                |
     @==========@             @==========@ S:OPEN                      |
+----| HDR_RCVD |             | HDR_SENT |------+                      |
|    @==========@             @==========@      |      R:HDR[!=S:HDR]  |
|   S:HDR |                      | R:HDR        |    +-----------------+
|         +--------+      +------+              |    |                 |
|                 \|/    \|/                   \|/   |                 |
|                @==========@               +-----------+ S:CLOSE      |
|                | HDR_EXCH |               | OPEN_PIPE |----+         |
|                @==========@               +-----------+    |         |
|           R:OPEN |      | S:OPEN              | R:HDR      |         |
|         +--------+      +------+      +-------+            |         |
|        \|/                    \|/    \|/                  \|/        |
|   @===========@             @===========@ S:CLOSE       +---------+  |
|   | OPEN_RCVD |             | OPEN_SENT |-----+         | OC_PIPE |--+
|   @===========@             @===========@     |         +---------+  |
|  S:OPEN |                      | R:OPEN      \|/           | R:HDR   |
|         |       @========@     |          +------------+   |         |
|         +------>| OPENED |<----+          | CLOSE_PIPE |<--+         |
|                 @========@                +------------+             |
|           R:CLOSE |    | S:CLOSE              | R:OPEN               |
|         +---------+    +-------+              |                      |
|        \|/                    \|/             |                      |
|   @============@            @============@    |                      |
|   | CLOSE_RCVD |            | CLOSE_SENT |<---+                      |
|   @============@            @============@                           |
| S:CLOSE |                      | R:CLOSE                             |
|         |         @=====@      |                                     |
|         +-------->| END |<-----+                                     |
|                   @=====@                                            |
|                     /|\                                              |
|    S:HDR[!=R:HDR]    |                R:HDR[!=S:HDR]                 |
+----------------------+-----------------------------------------------+

                     R:<CTRL> = Received <CTRL>
                     S:<CTRL> = Sent <CTRL>
```

State	Legal Sends	Legal Receives	Legal Connection Actions
START	HDR	HDR	
HDR_RCVD	HDR	OPEN	
HDR_SENT	OPEN	HDR	
HDR_EXCH	OPEN	OPEN	
OPEN_RCVD	OPEN	*	
OPEN_SENT	**	OPEN	
OPEN_PIPE	**	HDR	
CLOSE_PIPE	-	OPEN	TCP Close for Write
OC_PIPE	-	HDR	TCP Close for Write
OPENED	*	*	
CLOSE_RCVD	*	-	TCP Close for Read
CLOSE_SENT	-	*	TCP Close for Write
END	-	-	TCP Close

```
*  = any command or control
-  = no command or control
** = any command or control known a priori to conform to the
     peer's capabilities and limitations
```

3.8 Authentication

If authentication is required by a peer, it must announce supported authentication mechanisms using the sasl-server-mechanisms field of the connection.open control. The partner must then choose one of the supported mechanisms and initiate a sasl exchange.

SASL Exchange

```
SASL Client       SASL Server
================================
SASL-INIT     -->
              ...
              <-- SASL-CHALLENGE *
SASL-RESPONSE -->
              ...
              <-- SASL-OUTCOME
--------------------------------
  * Note that the SASL
    challenge/response step may
    occur zero or more times
    depending on the details of
    the SASL mechanism chosen.
```

The peer playing the role of the SASL Client and the peer playing the role of the SASL server may or may not correspond to the TCP client/server or the AMQP client/server. In fact, if mutual authentication is required, each peer will play the role of both the SASL Client and the SASL Server.

3.9 Connection Controls

3.9.1 Control: 0x0101 *(negotiate connection parameters)*

Signature: **open**(options: *map*, container-id: *str16*, hostname: *str16*, max-frame-size: *uint32*, channel-max: *uint16*, heartbeat-interval: *uint16*, sasl-server-mechanisms: *str16*, outgoing-locales: *str16*, incoming-locales: *str16*, peer-properties: *map*)

The open control MUST be the first frame sent in each direction on the connection. (Note that the connection header which is sent first on the connection is *not* a frame.) The fields indicate the capabilities and limitations of the sending peer.

Field Details:

options: *map* — options map **(optional)**

container-id: *str16* — the id of the source container **(required)**

hostname: *str16* — the name of the target host **(optional)**

The dns name of the host (either fully qualified or relative) to which the sending peer is connecting. It is not mandatory to provide the hostname. If no hostname is provided the receiving peer should select a default based on its own configuration.

max-frame-size: *uint32* — proposed maximum frame size **(optional)**

The largest frame size that the sending peer is able to accept on this connection. If this field is not set it means that the peer does not impose any specific limit. A peer MUST NOT send frames larger than its partner can handle. A peer that receives an oversized frame MUST close the connection with the framing-error close-code.
Both peers MUST accept frames of up to 4096 octets large.

channel-max: *uint16* — the maximum channel number that may be used on the connection **(required)**

The channel-max value is the highest channel number that may be used on the connection. This value plus one is the maximum number of sessions that may be simultaneously attached. A peer MUST not use channel numbers outside the range that its partner can handle. A peer that receives a channel number outside the supported range MUST close the connection with the framing-error close-code.

heartbeat-interval: *uint16* — proposed heartbeat interval **(optional)**

The proposed interval, in seconds, of the connection heartbeat desired by the sender. A value of zero means heartbeats are not supported. If the value is not set, the sender supports all heartbeat intervals. The heartbeat-interval established is the minimum of the two proposed heartbeat-intervals. If neither value is set, there is no heartbeat.

sasl-server-mechanisms: *str16* — supported sasl mechanisms **(multiple)**

A list of the sasl security mechanisms supported by the sending peer. If the sending peer does not require its partner to authenticate with it, this array may be empty or absent. The server mechanisms are ordered in decreasing level of preference.

outgoing-locales: *str16* — locales available for outgoing text **(multiple)**

A list of the locales that the peer supports for sending informational text. This includes connection close text, reject text, and session exception text. The default is the en_US locale. A peer MUST support at least the en_US locale. Since this value is always supported, it need not be supplied in the outgoing-locales array.

incoming-locales: *str16*	desired locales for incoming text in decreasing level of preference **(multiple)**

A list of locales that the sending peer permits for incoming informational text. This list is ordered in decreasing level of preference. The receiving partner will chose the first (most preferred) incoming locale from those which it supports. If none of the requested locales are supported, en_US will be chosen. Note that en_US need not be supplied in this list as it is always the fallback. A peer may determine which of the permitted incoming locales is chosen by examining the partner's supported locales as specified in the outgoing-locales field.

peer-properties: *map*	peer properties **(optional)**

The properties SHOULD contain at least these fields: "product", giving the name of the client product, "version", giving the name of the client version, "platform", giving the name of the operating system, "copyright", if appropriate, and "information", giving other general information.

Definition: MIN-MAX-FRAME-SIZE

Value: *4096* ***Description:*** *the minimum size (in bytes) of the maximum frame size*

During the initial connection negotiation, the two peers must agree upon a maximum frame size. This constant defines the minimum value to which the maximum frame size can be set. By defining this value, the peers can guarantee that they can send frames of up to this size until they have agreed a definitive maximum frame size for that connection.

3.9.2 Control: 0x0102 *(initiate sasl exchange)*

Signature: **sasl-init**(options: *map*, mechanism: *str16*, initial-response: *vbin32*)

Selects the sasl mechanism and provides the initial response if needed.

Field Details:

options: *map*	options map **(optional)**
mechanism: *str16*	selected security mechanism **(required)**

The name of the SASL mechanism used for the SASL exchange. If the selected mechanism is not supported by the receiving peer, it MUST close the connection with the authentication-failure close-code. Each peer MUST authenticate using the highest-level security profile it can handle from the list provided by the partner.

initial-response: *vbin32*	security response data **(optional)**

A block of opaque data passed to the security mechanism. The contents of this data are defined by the SASL security mechanism.

3.9.3 Control: 0x0103 *(security mechanism challenge)*

Signature: **sasl-challenge**(options: *map*, challenge: *vbin32*)

Send the SASL challenge data as defined by the SASL specification.

Field Details:

options: *map*	options map **(optional)**
challenge: *vbin32*	security challenge data **(required)**

Challenge information, a block of opaque binary data passed to the security mechanism.

3.9.4 Control: 0x0104 *(security mechanism response)*

Signature: **sasl-response**(options: *map*, response: *vbin32*)

Send the SASL response data as defined by the SASL specification.

Field Details:

options: *map*	options map **(optional)**
response: *vbin32*	security response data **(required)**

A block of opaque data passed to the security mechanism. The contents of this data are defined by the SASL security mechanism.

3.9.5 Control: 0x0105 *(indicates the outcome of the sasl dialog)*

Signature: **sasl-outcome**(options: *map*, code: *sasl-code*, additional-data: *vbin32*)

This control indicates the outcome of the SASL dialog.

Field Details:

options: *map*	options map **(optional)**
code: *sasl-code*	indicates the outcome of the sasl dialog **(optional)**

A reply-code indicating the outcome of the SASL dialog.

additional-data: *vbin32*	additional data as specified in RFC-4422 **(optional)**

The additional-data field carries additional data on successful authentication outcome as specified by the SASL specification (RFC-4422). If the authentication is unsuccessful, this field is not set.

3.9.6 sasl-code*: uint8 (codes to indicate the outcome of the sasl dialog)*

Valid values:

0 (ok)	Connection authentication succeeded.
1 (auth)	Connection authentication failed due to an unspecified problem with the supplied credentials.
2 (sys)	Connection authentication failed due to a system error.
3 (sys-perm)	Connection authentication failed due to a system error that is unlikely to be corrected without intervention.
4 (sys-temp)	Connection authentication failed due to a transient system error.

3.9.7 Control: 0x0106 *(signal a connection close)*

Signature: **close**(options: *map*, close-code: *close-code*, close-text: *str16*)

Sending a close signals that the sender will not be sending any more commands or controls on the connection. This control MUST be the last command or control written to a connection by the sender.

Field Details:

options: *map*	options map **(optional)**
close-code: *close-code*	connection close code **(required)**

A numeric code indicating the reason for the connection closure.

close-text: *str16*	connection close text **(optional)**

This text supplies any supplementary details not indicated by the connection close-code. This text can be logged as an aid to resolving issues.

3.9.8 close-code*: uint16 (codes used to indicate the reason for closure)*

Valid values:

200 (normal)	The connection closed normally.
320 (connection-forced)	An operator intervened to close the connection for some reason. The client may retry at some later date.
401 (authentication-failure)	The SASL authentication exchange failed.
402 (invalid-path)	The client tried to work with an unknown virtual host.
501 (framing-error)	A valid frame header cannot be formed from the incoming byte stream.

4 Session

An AMQP Session is a named dialog between two AMQP peers. Each participant maintains a Session Endpoint that stores the conversational state for that session. Session Endpoints are "attached" when they are associated with an open Connection. Session Endpoints may become detached deliberately, when a failure occurs in the network, or when a failure occurs at one of the endpoints. Session Endpoints may be configured to retain their state either temporarily or permanently when they become detached.

4.1 Naming a Session

Session names are supplied by the initiating peer, and MUST be globally unique among all open sessions. Once a session is cleanly closed, its may be reused. Session names are represented by opaque binary strings up to 65535 characters long. Example naming schemes include mechanically generated UUID-based names as well as stable, manually chosen URI based names. Session names are only exchanged during the attach and detach procedures, so there is no significant overhead to choosing large names.

4.1.1 session-name*: vbin16 (opaque session name)*

The session name uniquely identifies an interaction between two peers. It is globally unique among all open sessions. Once a session is cleanly closed, its name may be reused.

4.2 Establishing a Session

Sessions are established by creating a Session Endpoint, assigning it to an unused channel number, and sending an ATTACH carrying the state of the newly created Endpoint. The partner responds with an ATTACH carrying the state of the corresponding Endpoint, creating and/or mapping the Endpoint to an unused channel number if necessary. To avoid accidentally resuming an existing session, the initiating peer may optionally wish to verify that the corresponding Endpoint is newly created.

```
Endpoint                               Endpoint
=============================================================
ATTACH(name=...,       [CH3]--------->
       opening=1, ...)        +--[CH7] ATTACH(name=...,
                             /                opening=1, ...)
                            /
(1)                    <---+

                           ...

-------------------------------------------------------------

  (1) The initiating peer can at this point verify that the
      corresponding Endpoint is newly created.
```

4.3 Resuming a Session

Sessions are resumed by assigning the existing detached Session Endpoint to an unused channel number and sending an ATTACH carrying the state of the resuming Endpoint. The partner responds with an ATTACH carrying the state of the corresponding Endpoint, creating and/or mapping the Endpoint to an unused channel number if necessary. The resuming peer may wish to verify that the corresponding Endpoint is not newly created. This may be used to detect whether conversational state has been lost.

```
Endpoint                                      Endpoint
=============================================================
ATTACH(name=...,       [CH3]--------->
       opening=0, ...)        +--[CH7] ATTACH(name=...,
                             /                opening=0, ...)
                            /
(1)                    <---+

                             ...

-------------------------------------------------------------

  (1) The resuming peer can at this point verify that the
      corresponding Endpoint is not newly created.
```

4.4 Detaching a Session

Sessions become detached automatically when the connection is interrupted or closed, or when an error is encountered while executing a command. A Session is explicitly detached by sending a DETACH control. Once the DETACH is sent, no more commands or controls may be sent to the other Session Endpoint without first reattaching. The detaching peer must still process incoming commands and controls until the other Endpoint's DETACH is received.

```
Endpoint A                             Endpoint B
============================================================

                             ...

DETACH(name=...,       [CH3]-------->
       closing=0, ...)        +--[CH7] DETACH(name=...,
(1)                          /               closing=0, ...)
                            /          (2)
(3)                    <---+

                             ...

------------------------------------------------------------

  (1) At this point no more commands or controls may be
      sent to Endpoint B without first reattaching to it,
      but incoming commands and controls may still be
      received.

  (2) At this point Endpoint B is fully detached from the
      connection.

  (3) At this point Endpoint A is fully detached from the
      connection.
```

4.5 Closing a Session

Sessions are closed by setting the closing bit and detaching in the normal way. If an Endpoint sets the closing bit, it indicates that after the detach is confirmed, the Endpoint will be destroyed.

```
Endpoint A                             Endpoint B
=========================================================

                             ...

DETACH(name=...,       [CH3]--------->
       closing=1, ...)        +--[CH7] DETACH(name=...,
(1)                          /            closing=1, ...)
                            /          (2)
(3)                    <---+

                             ...

---------------------------------------------------------

  (1) At this point no more commands or controls may be
      sent to Endpoint B without first reattaching to
      it, but incoming commands and controls may still
      be received.

  (2) At this point Endpoint B is fully detached from
      the connection.

  (3) At this point Endpoint A is fully detached from
      the connection and can be destroyed.
```

4.6 Simultaneous Detach/Close

Due to the potentially asynchronous nature of Sessions, it is possible that both peers may simultaneously decide to detach and/or close the Session. If this should happen, it will appear to each peer as though their partner's spontaneously initiated DETACH is actually an answer to the peers initial DETACH control. One observable consequence of this occurrence is that if one Endpoint is closing and the other Endpoint is detaching, the values of the closing flag may not match. In this case, the closing Endpoint should reattach and verify that the detaching Endpoint was actually closed.

```
Endpoint A                                  Endpoint B
===============================================================

                              ...

DETACH(name=..., ...) [CH3]--+   +--[CH7] DETACH(name=..., ...)
(1)                           \ /          (2)
                               x
                              / \
(3)                   <------+   +------> (4)

                              ...

---------------------------------------------------------------

  (1) At this point no more commands or controls may be sent
      by A.

  (2) At this point no more commands or controls may be sent
      by B.

  (3) At this point Endpoint A is fully detached from the
      connection.

  (4) At this point Endpoint B is fully detached from the
      connection.
```

4.7 Session States

DETACHED: In the detached state, the session endpoint is not mapped to an open Connection. In this state an endpoint cannot send or receive commands and controls.

ATTACH_SENT: In the ATTACH_SENT state, the session endpoint is assigned an outgoing channel number, but there is no entry in the incoming channel map. In this state the endpoint may send commands and controls but cannot receive them.

ATTACH_RCVD: In the ATTACH_RCVD state, the session endpoint has an entry in the incoming channel map, but has not yet been assigned an outgoing channel number. The endpoint may receive commands and controls, but cannot send them.

ATTACHED: In the ATTACHED state, the session endpoint has both an outgoing channel number and an entry in the incoming channel map. The endpoint may send and receive commands and controls.

DETACH_SENT: In the DETACH_SENT state, the session endpoint has an entry in the incoming channel map, but is no longer assigned an outgoing channel number. The endpoint may receive commands and controls, but cannot send them.

DETACH_RCVD: In the DETACH_RCVD state, the session endpoint is assigned an outgoing channel number, but there is no entry in the incoming channel map. The endpoint may send commands and controls, but cannot receive them.

State Transitions

```
              DETACHED<----------------+
                 |                     |
         +-------+-------+             |
         |               |             |
S:ATTACH |               | R:ATTACH    |
         |               |             |
        \|/             \|/            |
    ATTACH_SENT     ATTACH_RCVD        |
         |               |             |
         |               |             |
R:ATTACH |               | S:ATTACH    |
         +-------+-------+             |
                 |                     |
                \|/                    |
              ATTACHED                 |
                 |                     |
         +-------+-------+             |
         |               |             |
S:DETACH |               | R:DETACH    |
         |               |             |
        \|/             \|/            |
    DETACH_SENT     DETACH_RCVD        |
         |               |             |
         |               |             |
R:DETACH |               | S:DETACH    |
         +-------+-------+             |
                 |                     |
                 |                     |
                 +---------------------+
```

4.8 Command Transport

A Session may be used as both a synchronous and an asynchronous command transport. By waiting for each command to be acknowledged prior to sending the next command, a Session becomes a fully synchronous command transport, and by permitting many unacknowledged commands a Session can become a fully asynchronous transport.

The diagram below depicts a snapshot of a Sender and Receiver in the process of sending and executing commands. The snapshot assumes perfect knowledge of the Sender and Receiver states at a given instant, and serves as an essential illustration of the different possible conditions that may occur when sending commands. For simplicity, the diagram shows only the state for a single Sender and Receiver pair.

Snapshot of a Session

```
              Endpoint                                 Endpoint
        +- - - - - - - - - +                      +- - - - - - - - +
        |     Receiver     |   <- - - - - - - -   |     Sender     |
        +- - - - - - - - - +                      +- - - - - - - - +
        +------------------+                      +----------------+
       /|      Sender      |\  --------------->  /|    Receiver    |\
      / +------------------+ \                  / +----------------+ \
     / /                    \ \                / /                  \ \
    / /                      \ \              / /                    \ \
   / /                        \ \            / /                      \ \
  / /                          \ \          / /                        \ \
 / /              replay buffer \ \        / /             input buffer \ \
+------------+---------------------+      +------------+-------------------+
| ... C2 C3  |  C4 C5 C6 C7 C8 C9  |      | ... C3 C4  |  C5 C6  |  __ __   |
+------------+---------------------+      +------------+-------------------+
 acknowledged           |     | |            executed  | received          |
                        +-----+ |                      +-------------------+
                           |    |                                |
                           +    +                             capacity
                          /      \
                   in-flight*   blocked

C1, C2, C3: executed by the receiver, and acknowledged at the sender
        C4: executed by the receiver, but not acknowledged at the sender
    C5, C6: received, but not yet executed by the receiver
    C7, C8: transmitted by the sender, but not yet received (in-flight)*
        C9: held by the sender awaiting capacity at the receiver

  * Note that should a network failure occur at this instant, the
    in-flight commands will be lost. In practice, neither the
    sender nor receiver knows explicitly which of the
    unacknowledged commands will be lost should a network failure
    occur.
```

4.9 Sender State

acknowledged: The id of the last command acknowledged. This marks the head of the replay buffer.

next-command: The id that will be assigned to the next command sent on the session. This marks the tail of the replay buffer. The id of any command in the replay buffer must fall between acknowledged and next-command: acknowledged < C.id < next-command for all C in the replay buffer

4.10 Receiver State

executed: The id of the last command executed. This marks the head of the input buffer.

received: The id of the last command received. This marks the end of the filled slots in the input buffer.

capacity: The capacity of the input buffer. The sum of executed and capacity is the id of the command that will fill the last slot in the input buffer.

Session endpoints periodically exchange "executed", "capacity", and "acknowledged". Each endpoint uses the knowledge of its partner's executed state to shrink the replay buffer for outgoing commands. Each endpoint uses the knowledge of its partners capacity to match its transmission rate to its partner's available receive capacity. Endpoints may use the knowledge of its partners acknowledged state to bound the set of incoming commands subject to replay at any given point.

```
Endpoint A                             Endpoint B
===========================================================
ATTACH(...)         [CH3]------------>
                              +--[CH7] ATTACH(...)
                             /
                            /
                    <------+

                            ...

STATE(acknowledged, [CH3]--+   +--[CH7] STATE(acknowledged,
      executed,             \ /               executed,
      capacity)              x                capacity)
                            / \
(1)                 <------+   +------> (1)

                            ...

-----------------------------------------------------------

  (1) Endpoints can shrink the replay buffer for
      outgoing commands, transmit additional commands,
      and bound the set of incoming commands subject to
      replay.
```

When Session endpoints (re)attach, they exchange "next-command", "received", and "capacity". Each endpoint then computes where to begin replay (if necessary), the id of the next incoming command, and when to block outgoing commands.

```
Endpoint                              Endpoint
===========================================================
ATTACH(next-command,  [CH3]---------> (1)
       received,             +--[CH7] ATTACH(next-command,
       capacity, ...)       /                received,
                           /                 capacity, ...)
(1)                   <---+

                            ...

-----------------------------------------------------------

  (1) Endpoints can establish where to begin replay, the
      expected id of the next incoming command, and the
      last outgoing command that may be sent before
      blocking.
```

4.11 Session Exceptions

Session exceptions occur when an error is encountered while processing a command. In this case the receiving peer MUST detach the session by issuing an outgoing detach control acknowledging the last successfully executed command and carrying information about the exceptional condition. Any "in-flight" commands received between the exceptional condition and the incoming detach MUST be thrown away.

Upon receiving any explicit detach, including those caused by exceptions, clients may choose to reattach and resume just as they would from an ordinary network failure. If the detach was caused by an exception, the client may choose, upon resume, to alter or omit any unexecuted commands, i.e. those following the exceptional condition. A client SHOULD examine the cause of failure and determine if it is likely to resolve itself before replaying unmodified commands.

```
Endpoint                      Endpoint
======================================================
COMMAND_1         ---------->
COMMAND_2         ---------->
COMMAND_3         ---+   +--- DETACH(executed=1,
                      \ /             exception="...")
                       x
                      / \
                  <--+   +--> *discarded*
DETACH            ---------->
                      ...
ATTACH            ---------->
                  <---------- ATTACH(executed=1, ...)
COMMAND_2'        ---------->
COMMAND_3         ---------->
                      ...
======================================================
```

4.12 Session Controls

4.12.1 Control: 0x0201 *(attach to the named session)*

Signature: **attach**(options: *map*, name: *session-name*, opening: *bit*, next-command: *sequence-no*, acknowledged: *sequence-no*, received: *sequence-no*, capacity: *uint32*, timeout: *uint32*, txn-mode: *txn-level*, txn-support: *txn-level*)

Indicate that a session endpoint has been attached to a connection. A session MUST NOT be mapped to more than one connection at a time.

Field Details:

options: *map* — options map **(optional)**

name: *session-name* — the session name **(required)**

The name of the session whose endpoint has been attached to the connection.

opening: *bit* — true iff the session endpoint has never been attached **(optional)**

This field, if set, indicates that the session endpoint is newly created and has never been attached to another endpoint.

next-command: *sequence-no* — **(optional)**

The id that will be assigned to the next outgoing command passed to the endpoint. For an opening endpoint this will be the id of the next command sent from the endpoint.

acknowledged: *sequence-no* — **(optional)**

The id of the last acknowledged command. This marks the head of the replay buffer. If the endpoint is newly created, this will be null.

received: *sequence-no* — the id of the last command received **(optional)**

The id of the last received command. This field MUST be set if and only if the session has received commands. This field indicates where replay will begin when resuming a session.

capacity: *uint32* — **(optional)**

This value should be given relative to received if the opening bit is false.

timeout: *uint32* — the session timeout **(optional)**

The session timeout indicates how long the endpoint will be kept when detached. If not set, the endpoint will be kept indefinitely.

txn-mode: *txn-level* — the desired txn-level **(optional)**

txn-support: *txn-level* — the maximum supported txn-level **(optional)**

4.12.2 Control: 0x0202 *(detach from the named session)*

Signature: **detach**(options: *map*, name: *session-name*, closing: *bit*, acknowledged: *sequence-no*, executed: *sequence-no*, exception: *exception*)

Indicates that the endpoint is being detached from the connection.

Field Details:

options: *map*	options map **(optional)**
name: *session-name*	the session name **(required)**

Identifies the detaching session.

closing: *bit*	**(optional)**

This field, if set, indicates that the session endpoint will be destroyed when fully detached.

acknowledged: *sequence-no*	**(optional)**

The id of the last acknowledged command. This marks the head of the replay buffer.

executed: *sequence-no*	**(optional)**

The id of the last executed command. This marks the head of the input buffer.

exception: *exception*	**(optional)**

If set, this field indicates that the session is being detached due to an exceptional condition. The value of the field should contain details on the cause of the exception.

4.12.3 exception struct (details of a session error)

This struct carries information on an exception which has occurred on the session. The command-id, when given, correlates the error to a specific command.

Field Details:

error-code: *error-code*	error code indicating the type of error **(required)**
command-id: *sequence-no*	exceptional command **(optional)**

The command-id of the command which caused the exception. If the exception was not caused by a specific command, this value is not set.

command-code: *uint8*	the class code of the command whose execution gave rise to the error (if appropriate) **(optional)**
field-index: *uint8*	index of the exceptional field **(optional)**

The zero based index of the exceptional field within the arguments to the exceptional command. If the exception was not caused by a specific field, this value is not set.

description: *str16*	descriptive text on the exception **(optional)**

The description provided is implementation defined, but MUST be in the language appropriate for the selected locale. The intention is that this description is suitable for logging or alerting output.

error-info: *map*	map to carry additional information about the error **(optional)**

4.12.4 error-code*: uint16 (error code used to identify the nature of an exception)*

Valid values:

4003	(unauthorized-access)	The client attempted to work with a server entity to which it has no access due to security settings.
4004	(not-found)	The client attempted to work with a server entity that does not exist.
4005	(resource-locked)	The client attempted to work with a server entity to which it has no access because another client is working with it.
4006	(precondition-failed)	The client requested a command that was not allowed because some precondition failed.
4008	(resource-deleted)	A server entity the client is working with has been deleted.
4009	(illegal-state)	The peer sent a command that is not permitted in the current state of the session.
4010	(transfer-limit-exceeded)	The peer sent more message transfers than currently allowed on the link.
5003	(command-invalid)	The command segments could not be decoded.
5006	(resource-limit-exceeded)	The client exceeded its resource allocation.
5030	(not-allowed)	The peer tried to use a command a manner that is inconsistent with the semantics defined in the specification.
5031	(illegal-argument)	The command argument is malformed, i.e. it does not fall within the specified domain. The illegal-argument exception can be raised on execution of any command which has domain valued fields.
5040	(not-implemented)	The peer tried to use functionality that is not implemented in its partner.
5041	(internal-error)	The peer could not complete the command because of an internal error. The peer may require intervention by an operator in order to resume normal operations.
5042	(invalid-argument)	An invalid argument was passed to a command, and the operation could not proceed. An invalid argument is not illegal (see illegal-argument), i.e. it matches the domain definition; however the particular value is invalid in this context.
6001	(xa-rbrollback)	The rollback was caused for an unspecified reason.
6002	(xa-rbtimeout)	A transaction branch took too long.
6003	(xa-heurhaz)	The transaction branch may have been heuristically completed.
6004	(xa-heurcom)	The transaction branch has been heuristically committed.
6005	(xa-heurrb)	The transaction branch has been heuristically rolled back.
6006	(xa-heurmix)	The transaction branch has been heuristically committed and rolled back.
6007	(xa-rdonly)	The transaction branch was read-only and has been committed.

4.12.5 Command: 0x0203 *(a command that does nothing)*

Signature: **noop**(options: *map*)

A command that does nothing.

Field Details:

options: *map*	options map **(optional)**

4.12.6 Command: 0x0204 *(executes an extended command)*

Signature: **execute**(options: *map*, spec: *str8*, name: *str8*, arguments: *map*)

Executes a command from an extended specification.

Field Details:

options: *map*	options map **(optional)**
spec: *str8*	an extended specification **(optional)**

Identifies the specification that defines the behavior for the command. This is typically a URI identifying a document that contains the formal definition for all the commands in the extended specification.

name: *str8*	the command name **(optional)**

Identifies the name of the command within the given specification.

arguments: *map*	the command arguments **(optional)**

Carries the command arguments as defined by the identified specification.

4.13 Transactional Sessions

A Transactional Session is one in which one of the two Applications using the Session behaves as a Transactional Resource, and the other behaves as a Transaction Controller. A Transactional Session must be in one of two modes: "local" or "distributed". In both modes, the Transaction Controller defines transactional units of work, and indicates whether each unit of work is a success or failure. On a Session with a txn-mode of "local", each successful unit of work is immediately committed as a complete transaction, and failed units of work are immediately rolled back. On a Session with a txn-mode of "distributed", each unit of work is given an xid and becomes part of a distributed transaction that is externally coordinated.

Applications capable of behaving as a Transactional Resource may advertise this when establishing a Session by setting the txn-support field of the `attach` control to "local" or "distributed". Applications that wish to behave as a Transaction Controller may indicate this and choose the mode for the Transactional Session by setting the txn-mode field of the `attach` control when establishing a Session. Only one end of a Session can be the Transaction Controller. It is an error for both Session endpoints to set the txn-mode field when when establishing a Session.

The transactional implications of each Command are determined by the semantics of the Transactional Resource. Some Commands sent from the Transaction Controller to the Transactional Resource may modify transactional state, while others may not. Likewise, some Commands sent from the Resource to the Controller may carry information about the transactional state of the Resource, while others may not. The state of the Command Transport itself is not transactional. A Commit or Rollback may negate or confirm the effects of a Command carried by the Transport, but it has no effect on the command-ids, replay-buffers, or other Transport related state that falls within the scope of the transaction.

4.13.1 txn-level*: uint8 (transaction level)*

Valid values:

1	(local)
2	(distributed)

4.13.2 Command: 0x0205 *(associate the current transactional work with a distributed transaction)*

Signature: **enlist**(options: *map*, xid: *xid*, join: *bit*, resume: *bit*)

Associates the current transactional unit of work with the distributed transaction identified by the supplied xid.

Field Details:

options: *map*	options map **(optional)**
xid: *xid*	Transaction xid **(required)**

Specifies the xid of the transaction branch in which to enlist.

join: *bit*	Join with existing xid flag **(optional)**

Indicate whether this is joining an already associated xid. Indicate that the enlist applies to joining a transaction previously seen.

resume: *bit*	Resume flag **(optional)**

Indicate that the enlist applies to resuming a suspended transaction branch.

Exceptions:

illegal-state	illegal-state *(4009)*

If the command is invoked in an improper context then the server MUST send a session exception.

already-known	not-allowed *(5030)*

If neither join nor resume is specified is specified and the transaction branch specified by xid has previously been seen then the server MUST raise an exception.

join-and-resume	not-allowed *(5030)*

If join and resume are specified then the server MUST raise an exception.

xa-rbrollback	xa-rbrollback *(6001)*

The broker marked the transaction branch rollback-only for an unspecified reason.

xa-rbtimeout	xa-rbtimeout *(6002)*

The work represented by this transaction branch took too long.

unknown-xid	not-allowed *(5030)*

If xid is already known by the broker then the server MUST raise an exception.

unsupported	not-implemented *(5040)*

If the broker does not support join the server MUST raise an exception.

4.13.3 xid record (dtx branch identifier)

An xid uniquely identifies a transaction branch.

Field Details:

format: *uint32*	implementation specific format code **(required)**
global-id: *vbin8*	global transaction id **(required)**
branch-id: *vbin8*	branch qualifier **(required)**

4.13.4 Command: 0x0206 *(mark transaction boundaries)*

Signature: **txn**(options: *map*, fail: *bit*, suspend: *bit*)

This command is called when the work done on behalf a transaction branch finishes or needs to be suspended. If neither fail nor suspend are specified then the portion of work has completed successfully. When a session is closed then the currently associated transaction branches MUST be marked rollback-only.

Field Details:

options: *map* — options map **(optional)**

fail: *bit* — Failure flag **(optional)**

If set, indicates that this portion of work has failed; otherwise this portion of work has completed successfully. An implementation MAY elect to roll a transaction back if this failure notification is received. Should an implementation elect to implement this behavior, and this bit is set, then then the transaction branch SHOULD be marked as rollback-only and the end result SHOULD have the xa-rbrollback status set.

suspend: *bit* — Temporary suspension flag **(optional)**

Indicates that the transaction branch is temporarily suspended in an incomplete state. The transaction context is in a suspended state and must be resumed via the enlist command with resume specified.

Exceptions:

illegal-state — illegal-state *(4009)*

If the command is invoked in an improper context (see class grammar) then the server MUST raise an exception.

suspend-and-fail — not-allowed *(5030)*

If both suspend and fail are specified then the server MUST raise an exception.

xa-rbrollback — xa-rbrollback *(6001)*

The broker marked the transaction branch rollback-only for an unspecified reason. If an implementation chooses to implement rollback-on-failure behavior, then this value should be selected if the dtx.end.fail bit was set.

xa-rbtimeout — xa-rbtimeout *(6002)*

The work represented by this transaction branch took too long.

5 Link

An AMQP network consists of message endpoints and intermediaries. Message endpoints are applications that produce or consume messages. Message intermediaries store and distribute messages as they travel between endpoints on the AMQP network.

```
  +----------+                +----------+
 /    MI1    \               /    MI2    \
+-------------+             +-------------+
|             | <---------> |             | <---------> ...
+-------------+             +-------------+
 /|\  /|\  /|\               /|\  /|\  /|\
  |    |    |                 |    |    |
  |    |    |                 |    |    |
  |    |    |                 |    |    |
 \|/  \|/  \|/               \|/  \|/  \|/
 ME1  ...  ME2               ME3  ...  ME4


      MI<n>: Message Intermediary <n>
      ME<n>: Message Endpoint <n>
```

Messages may be sent to and/or from an AMQP Node. A producer sends messages to a node; a consumer requests that some or all messages be sent from a node. Message intermediaries then determine how to distribute messages by matching producer messages to consumer interest.

```
                 +----------+                    +---------------------------+
                /    MI1    \ - - - - - - - - - | ME3 wants messages from A |
      M(to=A)  +-------------+  M(to=B)          |                           |
ME2 ---------> |             | <--------- ME1    | ME4 wants messages from A |
               +-------------+                   | matching criteria C       |
      M(to=A)    |    |    |                     |                           |
ME3 <------------+    |    |                     | MI2 wants messages from B |
                      |    |                     | and A                     |
      M(to=A) ? if C  |    |                     +---------------------------+
ME4 <-----------------+    |
                           |
    +----------------------+
    |                        +----------+        +---------------------------+
    |                       /    MI2    \ - - - | ME5 wants messages from B |
    | M(to=A), M(to=B)     +-------------+       |                           |
    +--------------------> |             |       | ME6 wants messages from A |
                           +-------------+       | and B                     |
      M(to=B)                |    |              +---------------------------+
ME5 <------------------------+    |
                                  |
      M(to=A), M(to=B)            |
ME6 <-----------------------------+

                  MI<n>: Message Intermediary <n>
                  ME<n>: Message Endpoint <n>
```

5.1 Links

Links establish which messages flow over a session, and control the rate of messages traveling to or from a given node. Each message transferred on a session must travel on an open link. Each link is established between two nodes -- a local node and a remote node. The link holds dynamic state associated with the transfer of messages between the two nodes. (Note that if a container establishes a session with itself, the local and remote nodes will actually be in the same container. A container may wish to provide a way to directly configure "internal" links in order to avoid creating a loopback session.)

Each link end is assigned a numeric handle used by the peer to refer to the link in all outgoing FLOW, DRAIN, and TRANSFER commands. In addition, each link has properties of directionality, durability, and distribution-mode. The directionality of a link is controlled by the direction field. Links may permit only incoming messages, only outgoing messages, or neither when closed. The durability of a link controls what happens to open links when a session is closed. The state of a durable link must be retained until that link is reopened. The state of a transient link is discarded. The distribution-mode of a link determines how messages from a node are distributed among its associated links. Messages from a node will be distributed to all associated links with a NON-DESTRUCTIVE distribution-mode, but to at most one associated link with a DESTRUCTIVE distribution mode.

```
+--------------------+                             +--------------------+
|      name: Link_1  |                             |      name: Link_1  |
|   durable: false   |                             |   durable: false   |
|    handle: i       |                             |    handle: j       |
|--------------------|                             |--------------------|
|     local: NODE_A  |                             |     local: NODE_B  |
|    remote: NODE_B  |<---+                   +--->|    remote: NODE_A  |
| direction: IN      |    |                   |    | direction: OUT     |
| dist-mode: -       |    |                   |    | dist-mode: -       |
+--------------------+    |                   |    +--------------------+
                          |                   |
                          |    +---------+    |
          ...           <---+--->| Session |<---+--->          ...
                          |    +---------+    |
                          |                   |
+--------------------+    |                   |    +--------------------+
|      name: Link_N  |    |                   |    |      name: Link_N  |
|   durable: true    |    |                   |    |   durable: true    |
|    handle: k       |    |                   |    |    handle: l       |
|--------------------|    |                   |    |--------------------|
|     local: -       |<---+                   +--->|     local: NODE_C  |
|    remote: NODE_C  |                             |    remote: -       |
| direction: OUT     |                             | direction: IN      |
| dist-mode: DEST    |                             | dist-mode: DEST    |
+--------------------+                             +--------------------+
```

5.2 Managing Links

The LINK, RELINK, and UNLINK commands are used to open, update, and close a link. An AMQP peer sends its partner the LINK command when a link end is created, and the UNLINK command when a link end is destroyed. The LINK command informs the partner of the state of the link end and maps it to a handle. The partner creates, the corresponding link end to match the peer, and responds with its own LINK command.

```
Peer                                     Partner
====================================================================
*create link end*
LINK(name=N, handle=1,       ----------> *create matching link end*
     direction=OUT, local=A        +---- LINK(name=N, handle=2,
     remote=B, ...)               /           direction=IN, local=B,
                             <---+            remote=A, ...)

                                 ...

RELINK(handle=1,             ---------->
       filter="...")         <---------- ACK(RELINK)

                                 ...

                                         *destroy link end*
*destroy link end*           <---------- UNLINK(handle=2)
UNLINK(handle=1)             ---------->
--------------------------------------------------------------------
```

5.3 Flow Control

Once established, the outgoing end of a link is subject to credit-based flow control of message transfers. The outgoing end of the link maintains a count of the total number of transfers it has sent since the link was opened, as well as a total transfer limit. If the sent count is the same or greater than the limit, it is illegal to send more transfer commands until the limit is increased. The incoming end of the link updates the transfer limit by sending FLOW commands. Flow control may be disabled entirely if the incoming end sends a FLOW command with an empty value for the limit. Although not required in all cases, it is often convenient for the incoming end of the link to maintain a received transfer count for use in determining an appropriate value for the updated transfer limit when sending a FLOW command. The sent transfer count, the received transfer count, and the transfer limit are all absolute values that must wraparound and compare according to RFC-1982 serial number arithmetic.

```
+----------------+                                 +----------------+
|     Link_1     |                                 |     Link_1     |
|----------------|                                 |----------------|
| direction=OUT  |<----+                     +---->| direction=IN   |
| transfers-sent |     |                     |     | transfers-rcvd |
| transfer-limit |     |                     |     |                |
+----------------+     |                     |     +----------------+
                       |                     |
                       |     +---------+     |
       ...        <----+---->| Session |<----+---->      ...
                       |     +---------+     |
                       |                     |
+----------------+     |                     |     +----------------+
|     Link_N     |     |                     |     |     Link_N     |
|----------------|     |                     |     |----------------|
| direction=IN   |<----+                     +---->| direction=OUT  |
| transfers-rcvd |                                 | transfers-sent |
|                |                                 | transfer-limit |
+----------------+                                 +----------------+
```

5.4 Controlling Outgoing Transfers

Each message transfer sent on a session increments the sent transfer count for the link on which it is sent. AMQP peers must not send transfers in excess of the current transfer limit of a link. If the transfer limit is reduced by the receiving peer when transfers are in-flight, the receiving peer may either handle the excess transfers normally or detach the session with a transfer-limit-exceeded error code. In the latter case if a session resume is attempted, the receiving peer should delay the session resume handshake until sufficient resources are available to handle the previously attempted in-flight transfers.

```
      +--------+
M1 -->| Link 1 |----+
      +--------+    |
                    |                 +---------+
           ...  ----+---- M1, M2 ---->| Session |---->
                    |                 +---------+
      +--------+    |
M2 -->| Link N |----+
      +--------+
          |
          |
          |
 if transfers-sent >= transfer-limit then block

              M<n>: Message Transfer<n>
```

5.5 Controlling Incoming Transfers

The FLOW and DRAIN commands control incoming transfers for a specific link. The FLOW command updates the transfer limit for the specified link. The DRAIN command tells the peer to send any immediately available transfers, and then set the transfer limit equal to the sent transfer count, thus stopping the link. Once the DRAIN command is complete, the link will be stopped, and any immediately available transfers will have been sent if permitted by the transfer limit prior to receiving the DRAIN. These commands may be used to provide a variety of different behaviors for receiving messages.

```
               +--------+
         M1 <--| Link 1 |<----+
               +--------+     |
                              |                      +---------+
                    ...    ----+----- FLOW/DRAIN ---->| Session |<---->
                              |                      +---------+
               +--------+     |
         M2 <--| Link N |<----+
               +--------+
                   |
                   |
                   |
       sync-get: FLOW(link=N, limit=rcvd+1)     ---->
      timed-get: FLOW(link=N, limit=rcvd+1),
                 *wait*,
                 DRAIN(link=N)                  ---->
   async-notify: FLOW(link=N, limit=rcvd+delta) ---->
                 - or -
                 FLOW(link=N, limit=null)       ---->
           stop: FLOW(link=N, limit=rcvd)       ---->

                           M<n>: Message Transfer<n>
```

5.6 Synchronous Get

A synchronous get of a message from a link is accomplished by incrementing the limit, sending a FLOW, and waiting indefinitely for a message to arrive.

```
Peer                                      Partner
==============================================================
                                ...
FLOW(handle=1,             ---------->
     limit=xfrs-rcvd+1)         +---- TRANSFER(handle=2, ...)
*block until xfr arrives*      /
                           <---+
                                ...
--------------------------------------------------------------
```

Synchronous get with a timeout is accomplished by incrementing the limit, waiting for the desired timeout, and then sending the DRAIN command. If the message does not arrive by the time execution of the DRAIN command is acknowledged, then the get times out.

```
    Peer                                      Partner
    ==============================================================
                                  ...
    FLOW(handle=1,            ---------->
         limit=xfrs-rcvd+1)
    *wait for timeout*
    DRAIN(handle=1)           ---+   +--> TRANSFER(handle=2, ...)
                                  \ /
                                   x
                                  / \
(1)                           <--+   +-->
(2)                           <---------- ACK(DRAIN)
                                  ...
    --------------------------------------------------------------
      (1) If a message is available within the timeout, it will
          arrive at this point.
      (2) If a message is not available within the timeout, the
          DRAIN command will ensure that the partner resets the
          transfer limit to the sent transfer count. Once
          peer gets acknowledgment of the execution of the DRAIN
          command, it can stop waiting for the message.
```

5.7 Asynchronous Notification

Asynchronous notification can be accomplished in two ways. If rate limiting of the messages from a given link is required, the receiver maintains a target delta between the transfer limit and the number of transfers received for that link. As messages arrive on the link, the actual delta decreases as the received count increases. When the actual delta falls below a threshold, a FLOW command is issued to increase the delta back to the desired target.

```
Peer                                    Partner
================================================================
                                ...
                            <---------- TRANSFER(handle=2, ...)
                            <---------- TRANSFER(handle=2, ...)
FLOW(handle=1,              ---+   +--- TRANSFER(handle=2, ...)
     limit=xfrs-rcvd+delta)     \ /
                                 x
                                / \
                            <--+   +-->
                            <---------- TRANSFER(handle=2, ...)
                            <---------- TRANSFER(handle=2, ...)
FLOW(handle=1,              ---+   +--- TRANSFER(handle=2, ...)
     limit=xfrs-rcvd+delta)     \ /
                                 x
                                / \
                            <--+   +-->
                                ...
----------------------------------------------------------------
  The incoming transfer rate for the link is limited by the
  rate at which the peer updates the transfer limit.
```

Alternatively, if there is no need to control the rate of incoming messages for the link, the receiver can send a FLOW command with an empty value for the limit field.

```
Peer                                    Partner
===============================================================
                                ...
FLOW(handle=1,              ---------->                        (1)
     limit=*empty*)               +---- TRANSFER(handle=2, ...)
                                 /
                            <---+
                            <---------- TRANSFER(handle=2, ...)
                            <---------- TRANSFER(handle=2, ...)
                                ...
---------------------------------------------------------------
  (1) Once the limit is disabled, the partner will transfer
      messages at whatever rate they become available.
```

5.8 Stopping a Link

Stopping the messages from a given link is accomplished by sending a FLOW command with the limit set to the received transfer count for that link. This guarantees that the limit will be less than or equal to the sent count at the other end of the link. Some transfers may be in-flight at the time the FLOW command is sent, so incoming transfers may still arrive on that link until execution of the FLOW command is acknowledged.

```
    Peer                                    Partner
    ================================================================
                                    ...
                                <---------- TRANSFER(handle=2, ...)
    FLOW(handle=1,              ---+   +--- TRANSFER(handle=2, ...)
         limit=xfrs-rcvd)           \ /
                                     x
                                    / \
(1)                             <--+   +-->
(2)                             <---------- ACK(FLOW)
                                    ...
    ----------------------------------------------------------------
      (1) In-flight transfers may still arrive until execution of
          the FLOW command is acknowledged.
      (2) At this point no further transfers will arrive.
```

5.9 Example: Outgoing Link

A message endpoint establishes an outgoing link by sending a LINK command with the link name, handle, remote node name, and the direction set to OUTGOING. The intermediary then creates or restores the corresponding link end and responds with its own LINK command followed by periodic FLOW commands to update the transfer limit for the link as capacity becomes available.

```
    Endpoint                              Intermediary
    ==============================================================
    LINK(name=N, handle=1,    ---------->
         remote=A,                  +---- LINK(name=N, handle=2,
         direction=OUT)            /           local=A,
                              <---+            direction=IN)
(1)                           <---------- FLOW(handle=2, limit=5)
    TRANSFER(handle=1, ...)   ---------->
    TRANSFER(handle=1, ...)   ---------->
    TRANSFER(handle=1, ...)   ---------->
    TRANSFER(handle=1, ...)   ---------->
(2) TRANSFER(handle=1, ...)   ---+   +--- FLOW(handle=2, limit=10)
                                  \ /
                                   x
                                  / \
(3)                           <--+   +-->
    TRANSFER(handle=1, ...)   ---------->
                                  ...
    --------------------------------------------------------------
      (1) The endpoint may now send up to 5 message transfers on
          the link.
      (2) The endpoint must block until further credit for the
          link is received.
      (3) The endpoint may now send an additional 5 message
          transfers on the link.
```

5.10 Example: Incoming Link

A message endpoint establishes an incoming link by sending a LINK command with the link name, handle, remote node name, and the direction set to INCOMING. The message intermediary creates or restores the corresponding link end, and sends its own LINK command. The message endpoint then sends FLOW commands to start and continue the flow of incoming message transfers for the link.

```
    Endpoint                                  Intermediary
    ==============================================================
    LINK(name=N, handle=1,    ---------->
         remote=A,                  +---- LINK(name=N, handle=2,
         direction=IN)             /           local=A,
                              <---+            direction=OUT)
(1) FLOW(handle=1, limit=5)   ---------->
                              <---------- TRANSFER(handle=2, ...)
                              <---------- TRANSFER(handle=2, ...)
                              <---------- TRANSFER(handle=2, ...)
                              <---------- TRANSFER(handle=2, ...)
(3) FLOW(handle=1, limit=10)  ---+   +--- TRANSFER(handle=2, ...) (2)
                                  \ /
                                   x
                                  / \
                              <--+   +-->                         (4)
                              <---------- TRANSFER(handle=2, ...)
                                   ...
    --------------------------------------------------------------
      (1) The endpoint may receive up to 5 message transfers on
          the link.
      (2) The intermediary may not send more message transfers
          until further credit for the link is received from the
          endpoint.
      (3) The endpoint must now be prepared to receive an
          additional 5 message transfers on the link.
      (4) The intermediary may now send up to 5 additional
          message transfers.
```

5.11 Example: Closing a Link

A link is closed by sending the UNLINK command with the link handle for the specified link. The peer will destroy the corresponding link end, and reply with its own UNLINK command.

```
    Endpoint                                 Intermediary
    ===========================================================
                                    ...
(1) FLOW(handle=1, ...)      <---------> FLOW(handle=2, ...)
    TRANSFER(handle=1, ...)  <---------> TRANSFER(handle=2, ...)
                                    ...
    UNLINK(handle=1)         ---------->
(2)                          <---------- UNLINK(handle=2)
    -----------------------------------------------------------
      (1) At this point the link is open. Depending on the
          directionality of the link, FLOW and TRANSFER
          commands may be sent freely in one or both
          directions.

      (2) At this point the link is now fully closed and no
          more FLOW or TRANSFER commands may be sent in
          either direction.
```

5.12 Link Commands

5.12.1 Command: 0x0301 *(carries the local state and desired remote state of a link)*

Signature: **link**(options: *map*, name: *str16*, durable: *bit*, handle: *handle*, create: *bit*, local: *node-name*, remote: *node-name*, direction: *direction*, distribution-mode: *distribution-mode*, filter: *vbin32*, filter-type: *str8*)

The link command is used to establish a link. An AMQP peer sends its partner the link command when a link end is created. The link command informs the partner of the new state of the link end. The partner creates, the corresponding link end to match the peer, and responds with its own link command confirming the state update.

Field Details:

options: *map* — options map **(optional)**

name: *str16* — the name of the link **(required)**

Link names are scoped to the client-id if durable, and scoped to the session-name if not durable. Link commands with the same name but differing durability refer to distinct links.
If the name refers to an existing link within the defined scope (either durable or non durable), and the local and remote node names are either not present or match the local and remote node names of the existing link, then the link end is updated to match the desired state specified in the link command. If the local and remote node names do **not** match the local and remote node names of the existing link, then the old link end is destroyed and a new link end is created in accordance with the link command.

durable: *bit* — indicates the scope and lifespan of the link **(optional)**

Durable links are scoped to the client-id and will last until explicitly closed. Non durable links are scoped to the session and may be closed explicitly, but will close automatically when the session terminates.

handle: *handle* — the link handle **(required)**

This field establishes the handle this endpoint will use to refer to the link in all subsequent outgoing commands.

create: *bit* — request creation of a remote node **(optional)**

The create flag, if set, indicates that the local peer would like the remote peer to generate a uniquely named remote node. The remote peer will supply the name of the generated node in the local field of the corresponding link command. If the create flag is set, the remote node name MUST be null. The generated node will exist until the link is destroyed.
The algorithm used to produce the node name from the link name, must produce repeatable results. If the link is durable, generating a node name from a given link name within a given client-id MUST always produce the same result. If the link is not durable, generating a node name from a given link name within a given session MUST always produce the same result. The generated node name SHOULD include the link name and session-name or client-id in some recognizable form for ease of traceability.

local: *node-name* — the name of the local node **(optional)**

remote: *node-name* — the name of the remote node **(optional)**

direction: *direction* — direction of the link **(optional)**

distribution-mode: *distribution-mode* — the distribution mode of the link **(optional)**

If set, the distribution-mode field indicates the distribution mode used for outgoing messages.

filter: *vbin32*	a predicate to filter the messages admitted onto the link **(optional)**
filter-type: *str8*	the type of the filter **(default: SQL-EXPRESSION)**

5.12.2 Command: 0x0302 *(re-establish link parameters)*

Signature: **relink**(options: *map*, handle: *handle*, filter: *vbin32*, filter-type: *str8*)

Re-establish link with new parameters.

Field Details:

options: *map*	options map **(optional)**
handle: *handle*	identifies the link **(optional)**
filter: *vbin32*	a predicate to filter the messages admitted onto the link **(optional)**
filter-type: *str8*	the type of the filter **(default: SQL-EXPRESSION)**

5.12.3 Command: 0x0303 *(close the link)*

Signature: **unlink**(options: *map*, handle: *handle*)

Close the link and un-map the handle.

Field Details:

options: *map*	options map **(optional)**
handle: *handle*	identifies the link **(optional)**

5.12.4 node-name*: vbin16 (name of the source or destination for a message)*

Specifies the name for a source or destination node to which messages are to be transferred to or from. Node names are expected to be human readable, but are intentionally considered opaque. The format of a node name is not defined by this specification.

5.12.5 handle*: uint32 (the handle of a link)*

An alias established by the link command and subsequently used by endpoints as a shorthand to refer to the link in all outgoing commands. The two endpoints may potentially use different handles to refer to the same link. Link handles may be reused once a link is closed for both send and receive.

5.12.6 direction*: uint8 (link direction)*

Valid values:

0 (incoming)

1 (outgoing)

5.12.7 distribution-mode*: uint32 (link distribution policy)*

Policies for distributing messages when multiple links are connected to the same node.

Valid values:

1 (destructive)	once successfully transferred over the link, the message will no longer be available to other links from the same node
2 (non-destructive)	once successfully transferred over the link, the message is still available for other links from the same node

5.12.8 Command: 0x0304 *(update the transfer limit for a link)*

Signature: **flow**(options: *map*, handle: *handle*, limit: *sequence-no*)

This command updates the transfer-limit for the specified link.

Field Details:

options: *map*	options map **(optional)**
handle: *handle*	the link handle **(required)**

Identifies the link whose transfer limit is to be updated.

limit: *sequence-no*	the link transfer limit **(optional)**

The updated value for the transfer-limit. This is the limit beyond which the sent transfer count for the link may not exceed. This is an absolute number and must wraparound and compare according to RFC-1982 serial number arithmetic. If this is not set, there is no limit and transfers may be sent until a limit is imposed.

5.12.9 Command: 0x0305 *(drain the link of immediately available transfers and stop it)*

Signature: **drain**(options: *map*, handle: *handle*)

This command causes any immediately available message transfers to be sent up to the pre-existing transfer limit. If the number of immediately available message transfers is insufficient to reach the pre-existing transfer limit, the transfer limit is reset to the sent transfer count. When this command completes, the transfer limit will always equal the sent transfer count.

Field Details:

options: *map*	options map **(optional)**
handle: *handle*	the link handle **(required)**

Identifies the link to be drained.

5.13 Transfer States

START:	The message has yet to be sent to the recipient.
TRANSFERRING:	The message has been partially sent to the recipient, but is not yet acquired.
ACQUIRED:	The message has been sent to and acquired by the recipient.
PARKED:	The message is held by the outgoing end of the link until unparked.
END:	The transfer is complete.

State Transitions

```
                            XFR(M=1,A=0)
                     START---------------+
                       |                 |
                       |                 |
                       |                 |      +------+
                       |                 |      |      |
                       |                \|/    \|/     | XFR(M=1,A=0)
          XFR(M=0,A=0) |               TRANSFERRING----+
                       |                 |      |
                       |    XFR(M=0,A=0) |      | XFR(M=0,A=1)
                       |   +-------------+      |
                       |   |                    |
                      \|/ \|/       PARK        |
                     ACQUIRED------------+      |
                       |                 |      |
                       |                 |      |
                       |                 |      |
                       |                 |      |
                       |                \|/     |
ACK / REJECT / RELEASE |               PARKED   |
                       |                 |      |
                       |                 |      |
                       |                 |      |
                       |                 |      |
                      \|/                |      |
                      END<---------------+      |
                      /|\         UNPARK        |
                       |                        |
                       +------------------------+


      Key: XFR(M=?,A=?) --> TRANSFER(more=?, aborted=?)
```

5.14 Transfer Commands

5.14.1 Command: 0x0306 *(transfer a message)*

Signature: **transfer**(options: *map*, handle: *handle*, more: *bit*, aborted: *bit*, redelivered: *bit*, fragment-offset: *uint64*, transmit-time: *timestamp*, ttl: *uint64*, header: *header*, payload: *vbin32*, footer: *footer*)

The transfer command is used to send messages across a link. Messages may be carried by a single transfer command up to the maximum negotiated frame size for the connection. Larger messages may be split across several consecutive transfer commands.

Field Details:

options: *map*	options map **(optional)**
handle: *handle*	**(required)**

Specifies the link on which the message is transferred.

more: *bit*	indicates that the message has more content **(optional)**
aborted: *bit*	indicates that the message is aborted **(optional)**

Aborted messages should be discarded by the recipient.

redelivered: *bit*	indicates possible duplication **(optional)**

This flag indicates that the message may be a duplicate. When delivered on an exclusive link the redelivered flag is set to true if:

the message when originally sent to the node had the redelivered flag set to true

the message has previously been sent from this node over an exclusive link to a session which closed without the message being acknowledged

the message has previously been sent from this node over an exclusive link and was subsequently released with the mark-redelivered field set to true

When delivered on a shared link the redelivered flag is set to true if any of the above conditions are met, or:

the message has previously been sent over the same shared link and subsequently released with the mark redelivered flag set to true

the shared link is durable, and the message was previously sent over this link on a session which was closed before the message was acknowledged

fragment-offset: *uint64*	the payload offset within the message **(optional)**
transmit-time: *timestamp*	the time of message transmit **(optional)**

The point in time at which the sender considers the message to be transmitted. The ttl field, if set by the sender, is relative to this point in time.

ttl: *uint64*	time to live in ms **(optional)**

Duration in milliseconds for which the message should be considered "live". If this is set then a message expiration time will be computed based on the transmit-time plus this value. Messages that live longer than their expiration time will be discarded (or dead lettered). If the transmit-time is not set, then the expiration is computed relative to the message arrival time.

header: *header*	message header **(optional)**
payload: *vbin32*	message payload **(optional)**
footer: *footer*	message footer **(optional)**

5.14.2 header struct (transport headers for a message)

The header struct carries information about the transfer of a message over a specific link.

Field Details:

durable: *bit* — specify durability requirements **(optional)**

Durable messages MUST NOT be lost even if an intermediary is unexpectedly terminated and restarted.

priority: *uint8* — relative message priority **(optional)**

This field contains the relative message priority. Higher numbers indicate higher priority messages. Messages with higher priorities MAY be delivered before those with lower priorities.
An AMQP intermediary implementing distinct priority levels MUST do so in the following manner:
If n distinct priorities are implemented and n is less than 10 -- priorities 0 to (5 - ceiling(n/2)) MUST be treated equivalently and MUST be the lowest effective priority. The priorities (4 + floor(n/2)) and above MUST be treated equivalently and MUST be the highest effective priority. The priorities (5 - ceiling(n/2)) to (4 + floor(n/2)) inclusive MUST be treated as distinct priorities.
If n distinct priorities are implemented and n is 10 or greater -- priorities 0 to (n - 1) MUST be distinct, and priorities n and above MUST be equivalent to priority (n - 1).
Thus, for example, if 2 distinct priorities are implemented, then levels 0 to 4 are equivalent, and levels 5 to 9 are equivalent and levels 4 and 5 are distinct. If 3 distinct priorities are implements the 0 to 3 are equivalent, 5 to 9 are equivalent and 3, 4 and 5 are distinct.
This scheme ensures that if two priorities are distinct for a server which implements m separate priority levels they are also distinct for a server which implements n different priority levels where n > m.

format-code: *uint32* — indicates the format of the message **(optional)**

5.14.3 footer struct (transport footers for a message)

TODO: import definitions from security SIG

5.14.4 Command: 0x0309 *(reject message transfers)*

Signature: **reject**(options: *map*, first: *sequence-no*, last: *sequence-no*, reject-properties: *map*)

The reject command is used to indicate that incoming messages are invalid and therefore unprocessable. Any message whose first incoming transfer falls within the specified incoming command range is considered rejected. Continuation transfers within the rejection range are ignored. For this command to have any effect, it MUST be sent before the identified transfers are acknowledged. If an attempt to transfer a message results in a reject from the recipient, the sender should add the supplied reject-properties to the message header, and make the message available at an alternative node (e.g. a dead letter queue).

Field Details:

options: *map* — options map **(optional)**

first: *sequence-no* — **(required)**

The start of the incoming command range.

last: *sequence-no* — **(optional)**

The end of the incoming command range. If not set then this is taken to be the same as first.

reject-properties: *map* — **(optional)**

The map supplied in this field will be placed in any rejected message headers under the key "reject-properties".

5.14.5 Command: 0x030a *(release messages)*

Signature: **release**(options: *map*, first: *sequence-no*, last: *sequence-no*, mark-redelivered: *bit*)

The release command is used to indicate that the specified range of incoming transfers will never be processed. Any continuation transfers within the specified range are ignored. For this command to have any effect, it MUST be sent before the identified transfers are acknowledged. The messages carried by the released transfer commands become available to the sender for future transfer to this or other sessions. Messages that have been released MAY subsequently be delivered out of order. Implementations SHOULD ensure that released messages keep their position with respect to undelivered messages of the same priority.

Field Details:

options: *map*	options map **(optional)**
first: *sequence-no*	**(required)**

The start of the incoming command range.

last: *sequence-no*	**(optional)**

The end of the incoming command range. If not set then this is taken to be the same as first.

mark-redelivered: *bit*	mark the released messages as redelivered **(optional)**

If the mark-redelivered flag is set, any messages released by this command MUST have the redelivered flag set according to the semantics specified in the delivery-properties definition.

5.14.6 Command: 0x030b *(park messages)*

Signature: **park**(options: *map*, first: *sequence-no*, last: *sequence-no*)

The park command is used to indicate that the specified range of incoming transfers cannot be processed at this time, but that other (future) transfers will continue to be processed, and therefore the indicated messages should be held on the link until unparked. Once a message has been parked it can no longer be released or rejected. Any continuation transfers within the specified range are ignored. For this command to have any effect, it MUST be sent before the identified transfers are acknowledged.

Field Details:

options: *map*	options map **(optional)**
first: *sequence-no*	**(required)**

The start of the incoming command range.

last: *sequence-no*	**(optional)**

The end of the incoming command range. If not set then this is taken to be the same as first.

5.14.7 Command: 0x030c *(unpark messages)*

Signature: **unpark**(options: *map*, mode: *unpark-mode*, first: *sequence-no*, last: *sequence-no*)

The unpark command is used to indicate that the specified range of incoming transfers should no longer be held on the link. The unpark-mode defines what should be done with the held messages. They may be acknowledged, released back to the queue, rejected, or resent on the same link. Any continuation transfers within the specified range are ignored.

Field Details:

options: *map*	options map **(optional)**
mode: *unpark-mode*	**(required)**

One of acknowledge, reject, release, resend. If resend is chosen for a message whose link has subsequently been deleted, the message will be released instead of resent.

first: *sequence-no*	**(required)**

The start of the incoming command range.

last: *sequence-no*	**(optional)**

The end of the incoming command range. If not set then this is taken to be the same as first.

5.14.8 unpark-mode*: uint8 (unpark behaviors)*

Valid values:

0	(acknowledge)
1	(reject)
2	(release)
3	(resend)

5.15 Message Format

5.15.1 message-properties struct (immutable properties of the message)

Message properties carry information about the message.

Field Details:

message-id: *vbin16*	application message identifier **(optional)**

Message-id is an optional property which uniquely identifies a message within the message system. The message producer is usually responsible for setting the message-id in such a way that it is assured to be globally unique. The server MAY discard a message as a duplicate if the value of the message-id matches that of a previously received message sent to the same node.

user-id: *vbin16* creating user id **(optional)**

The identity of the user responsible for producing the message. The client sets this value, and it MAY be authenticated by intermediaries.

to: *node-name* the name of the node the message is destined for **(optional)**

The to field identifies the node that is the intended destination of the message. On any given transfer this may not be the node at the receiving end of the link.

reply-to: *node-name* the node to send replies to **(optional)**

The name of the node to send replies to.

correlation-id: *vbin16* application correlation identifier **(optional)**

This is a client-specific id that may be used to mark or identify messages between clients. The server ignores this field.

content-length: *uint64* length of the combined payload in bytes **(optional)**

The total size in octets of the combined payload of all message.transfer commands that together make the message.

content-type: *str8* MIME content type **(optional)**

The RFC-2046 MIME type for the message content (such as "text/plain"). This is set by the originating client. As per RFC-2046 this may contain a charset parameter defining the character encoding used: e.g. 'text/plain; charset="utf-8"'.

properties: *map* application defined message properties **(optional)**

6 Basic-types

Each AMQP type defines a format for encoding a particular kind of data. Additionally, most AMQP types are assigned a unique code that functions as a discriminator when more than one type may be encoded in a given position.

AMQP types broadly fall into two categories: fixed-width and variable-width. Variable-width types are always prefixed by a byte count of the encoded size, excluding the bytes required for the byte count itself. Unless otherwise specified, AMQP uses network byte order for all numeric values.

A type code is a single octet which may hold 256 distinct values. Ranges of types are mapped to specific sizes of data so that an implementation can easily skip over any data types not natively supported.

```
Code         Category        Format
==============================================================================
0x00 - 0x0F  Fixed Width     One octet of data.
0x10 - 0x1F  Fixed Width     Two octets of data.
0x20 - 0x2F  Fixed Width     Four octets of data.
0x30 - 0x3F  Fixed Width     Eight octets of data.
0x40 - 0x4F  Fixed Width     Sixteen octets of data.
0x50 - 0x5F  Fixed Width     Thirty-two octets of data.
0x60 - 0x6F  Fixed Width     Sixty-four octets of data.
0x70 - 0x7F  Fixed Width     One hundred twenty-eight octets of data.
0x80 - 0x8F  Variable Width  One octet of size, 0-255 octets of data.
0x90 - 0x9F  Variable Width  Two octets of size, 0-65535 octets of data.
0xA0 - 0xAF  Variable Width  Four octets of size, 0-4294967295 octets of data.
0xB1 - 0xBF  Reserved
0xC0 - 0xCF  Fixed Width     Five octets of data.
0xD0 - 0xDF  Fixed Width     Nine octets of data.
0xE0 - 0xEF  Reserved
0xF0 - 0xFF  Fixed Width     Zero octets of data.
```

The particular type code ranges were chosen with the following rationale in mind:

```
                Bit:  7    6    5    4    3    2    1    0
                ------------------------------------------
                      0  |   fix-exp    |     subtype
                      1    0  | var-exp |     subtype
                      1    1  | fix-odd |     subtype
                ------------------------------------------

          fix-exp = log2(size of fixed width type)
          var-exp = log2(size of size of variable width type) (Note: 11 is reserved)
          fix-odd = 00, for 5-byte fixed width
                    01, for 9-byte fixed width
                    10, reserved
                    11, for 0-byte fixed width
```

6.1.1 Fixed width types:

Name	Code	Width in Octets	Description
bin8	0x00	1	octet of unspecified encoding
int8	0x01	1	8-bit signed integral value (-128 - 127)
uint8	0x02	1	8-bit unsigned integral value (0 - 255)
char	0x04	1	an iso-8859-15 character
boolean	0x08	1	boolean value (zero represents false, nonzero represents true)
bin16	0x10	2	two octets of unspecified binary encoding
int16	0x11	2	16-bit signed integral value
uint16	0x12	2	16-bit unsigned integer
bin32	0x20	4	four octets of unspecified binary encoding
int32	0x21	4	32-bit signed integral value
uint32	0x22	4	32-bit unsigned integral value
float	0x23	4	single precision IEEE 754 32-bit floating point
char-utf32	0x27	4	single unicode character in UTF-32 encoding
sequence-no	0x28	4	serial number defined in RFC-1982
bin64	0x30	8	eight octets of unspecified binary encoding
int64	0x31	8	64-bit signed integral value
uint64	0x32	8	64-bit unsigned integral value
double	0x33	8	double precision IEEE 754 floating point
timestamp	0x38	8	timestamp in 64 bit POSIX time_t format
bin128	0x40	16	sixteen octets of unspecified binary encoding
uuid	0x48	16	UUID (RFC-4122 section 4.1.2) - 16 octets
bin256	0x50	32	thirty two octets of unspecified binary encoding
bin512	0x60	64	sixty four octets of unspecified binary encoding
bin1024	0x70	128	one hundred and twenty eight octets of unspecified binary encoding
bin40	0xc0	5	five octets of unspecified binary encoding
dec32	0xc8	5	32-bit decimal value (e.g. for use in financial values)
bin72	0xd0	9	nine octets of unspecified binary encoding
dec64	0xd8	9	64-bit decimal value (e.g. for use in financial values)
void	0xf0	0	the void type
bit	0xf1	0	presence indicator

6.1.2 Variable width types

Variable width types consist of a number of octets which represent an unsgigned integral size; followed by the given number of octets. The size field should be read as if it were a uint8, if there is one size octet, as a uint16 if there are two size octets, a unit32 if there are four size octets, and so on.

Name	Code	Size Octets	Description
vbin8	0x80	1	up to 255 octets of opaque binary data
str8-latin	0x84	1	up to 255 iso-8859-15 characters
str8	0x85	1	up to 255 octets worth of UTF-8 unicode
str8-utf16	0x86	1	up to 255 octets worth of UTF-16 unicode
vbin16	0x90	2	up to 65535 octets of opaque binary data
str16-latin	0x94	2	up to 65535 iso-8859-15 characters
str16	0x95	2	up to 65535 octets worth of UTF-8 unicode
str16-utf16	0x96	2	up to 65535 octets worth of UTF-16 unicode
vbin32	0xa0	4	up to 4294967295 octets of opaque binary data
map	0xa8	4	a mapping of keys to typed values
list	0xa9	4	a series of consecutive type-value pairs
array	0xaa	4	a defined length collection of values of a single type
struct	0xab	4	any struct

6.2 Fixed Width (1 octet)

6.2.1 Type: bin8

The bin8 type consists of exactly one octet of opaque binary data.

Wire Format

```
   1 OCTET
+----------+
|   bin8   |
+----------+
```

6.2.2 Type: int8

The int8 type is a signed integral value encoded using an 8-bit two's complement representation.

Wire Format

```
   1 OCTET
+----------+
|   int8   |
+----------+
```

6.2.3 Type: uint8

The uint8 type is an 8-bit unsigned integral value.

Wire Format

```
  1 OCTET
+---------+
|  uint8  |
+---------+
```

6.2.4 Type: char

The char type encodes a single character from the iso-8859-15 character set.

Wire Format

```
  1 OCTET
+----------+
|   char   |
+----------+
```

6.2.5 Type: boolean

The boolean type is a single octet that encodes a true or false value. If the octet is zero, then the boolean is false. Any other value represents true.

Wire Format

```
  1 OCTET
+---------+
| boolean |
+---------+
```

6.3 Fixed Width (2 octets)

6.3.1 Type: bin16

The bin16 type consists of two consecutive octets of opaque binary data.

Wire Format

```
  1 OCTET     1 OCTET
+-----------+-----------+
| octet-one | octet-two |
+-----------+-----------+
```

6.3.2 Type: int16

The int16 type is a signed integral value encoded using a 16-bit two's complement representation in network byte order.

Wire Format

```
  1 OCTET    1 OCTET
+-----------+----------+
| high-byte | low-byte |
+-----------+----------+
```

6.3.3 Type: uint16

The uint16 type is a 16-bit unsigned integral value encoded in network byte order.

Wire Format

```
  1 OCTET    1 OCTET
+-----------+----------+
| high-byte | low-byte |
+-----------+----------+
```

6.4 Fixed Width (4 octets)

6.4.1 Type: bin32

The bin32 type consists of 4 consecutive octets of opaque binary data.

Wire Format

```
  1 OCTET     1 OCTET      1 OCTET      1 OCTET
+-----------+-----------+-------------+------------+
| octet-one | octet-two | octet-three | octet-four |
+-----------+-----------+-------------+------------+
```

6.4.2 Type: int32

The int32 type is a signed integral value encoded using a 32-bit two's complement representation in network byte order.

Wire Format

```
  1 OCTET     1 OCTET     1 OCTET    1 OCTET
+-----------+------------+----------+----------+
| byte-four | byte-three | byte-two | byte-one |
+-----------+------------+----------+----------+
    MSB                                  LSB
```

6.4.3 Type: uint32

The uint32 type is a 32-bit unsigned integral value encoded in network byte order.

Wire Format

```
  1 OCTET     1 OCTET     1 OCTET    1 OCTET
+-----------+------------+----------+----------+
| byte-four | byte-three | byte-two | byte-one |
+-----------+------------+----------+----------+
    MSB                                  LSB
```

6.4.4 Type: float

The float type encodes a single precision 32-bit floating point number. The format and operations are defined by the IEEE 754 standard for 32-bit floating point numbers.

Wire Format

```
              4 OCTETs
    +-----------------------+
    |         float         |
    +-----------------------+
      IEEE 754 32-bit float
```

6.4.5 Type: char-utf32

The char-utf32 type consists of a single unicode character in the UTF-32 encoding.

Wire Format

```
         4 OCTETs
    +-------------------+
    |    char-utf32     |
    +-------------------+
      UTF-32 character
```

6.4.6 Type: sequence-no

The sequence-no type encodes, in network byte order, a serial number as defined in RFC-1982. The arithmetic, operators, and ranges for numbers of this type are defined by RFC-1982.

Wire Format

```
               4 OCTETs
    +-------------------------+
    |       sequence-no       |
    +-------------------------+
      RFC-1982 serial number
```

6.5 Fixed Width (8 octets)

6.5.1 Type: bin64

The bin64 type consists of eight consecutive octets of opaque binary data.

Wire Format

```
   1 OCTET     1 OCTET            1 OCTET       1 OCTET
+-----------+-----------+-----+-------------+-------------+
| octet-one | octet-two | ... | octet-seven | octet-eight |
+-----------+-----------+-----+-------------+-------------+
```

6.5.2 Type: int64

The int64 type is a signed integral value encoded using a 64-bit two's complement representation in network byte order.

Wire Format

```
   1 OCTET      1 OCTET            1 OCTET    1 OCTET
+------------+------------+-----+----------+----------+
| byte-eight | byte-seven | ... | byte-two | byte-one |
+------------+------------+-----+----------+----------+
    MSB                                        LSB
```

6.5.3 Type: uint64

The uint64 type is a 64-bit unsigned integral value encoded in network byte order.

Wire Format

```
   1 OCTET      1 OCTET            1 OCTET    1 OCTET
+------------+------------+-----+----------+----------+
| byte-eight | byte-seven | ... | byte-two | byte-one |
+------------+------------+-----+----------+----------+
    MSB                                        LSB
```

6.5.4 Type: double

The double type encodes a double precision 64-bit floating point number. The format and operations are defined by the IEEE 754 standard for 64-bit double precision floating point numbers.

Wire Format

```
        8 OCTETs
+-----------------------+
|        double         |
+-----------------------+
  IEEE 754 64-bit float
```

6.5.5 Type: timestamp

The timestamp type encodes a point in time using the 64 bit POSIX time_t format. This is a signed 64 bit number representing milliseconds since the epoch.

Wire Format

```
       8 OCTETs
+----------------------+
|      timestamp       |
+----------------------+
  posix time_t format
```

6.6 Fixed Width (16 octets)

6.6.1 Type: bin128

The bin128 type consists of 16 consecutive octets of opaque binary data.

Wire Format

```
   1 OCTET     1 OCTET              1 OCTET         1 OCTET
+-----------+-----------+-----+---------------+----------------+
| octet-one | octet-two | ... | octet-fifteen | octet-sixteen |
+-----------+-----------+-----+---------------+----------------+
```

6.6.2 Type: uuid

The uuid type encodes a universally unique id as defined by RFC-4122. The format and operations for this type can be found in section 4.1.2 of RFC-4122.

Wire Format

```
    16 OCTETs
+---------------+
|     uuid      |
+---------------+
  RFC-4122 UUID
```

6.7 Fixed Width (32 octets)

6.7.1 Type: bin256

The bin256 type consists of thirty two consecutive octets of opaque binary data.

Wire Format

```
   1 OCTET     1 OCTET               1 OCTET            1 OCTET
+-----------+-----------+-----+------------------+------------------+
| octet-one | octet-two | ... | octet-thirty-one | octet-thirty-two |
+-----------+-----------+-----+------------------+------------------+
```

6.8 Fixed Width (64 octets)

6.8.1 Type: bin512

The bin512 type consists of sixty four consecutive octets of opaque binary data.

Wire Format

```
   1 OCTET     1 OCTET                1 OCTET             1 OCTET
+-----------+-----------+-----+-------------------+------------------+
| octet-one | octet-two | ... | octet-sixty-three | octet-sixty-four |
+-----------+-----------+-----+-------------------+------------------+
```

6.9 Fixed Width (128 octets)

6.9.1 Type: bin1024

The bin1024 type consists of one hundred and twenty eight octets of opaque binary data.

Wire Format

```
  1 OCTET     1 OCTET                 1 OCTET                  1 OCTET
+-----------+-----------+-----+------------------------+------------------------+
| octet-one | octet-two | ... | octet-one-twenty-seven | octet-one-twenty-eight |
+-----------+-----------+-----+------------------------+------------------------+
```

6.10 Variable Width (1 octet size)

6.10.1 Type: vbin8

The vbin8 type encodes up to 255 octets of opaque binary data. The number of octets is first encoded as an 8-bit unsigned integral value. This is followed by the actual data.

Wire Format

```
  1 OCTET   size OCTETs
+---------+-------------+
|  size   |   octets    |
+---------+-------------+
   uint8
```

6.10.2 Type: str8-latin

The str8-latin type encodes up to 255 octets of iso-8859-15 characters. The number of octets is first encoded as an 8-bit unsigned integral value. This is followed by the actual characters.

Wire Format

```
  1 OCTET        size OCTETs
+---------+------------------------+
|  size   |       characters       |
+---------+------------------------+
   uint8    iso-8859-15 characters
```

6.10.3 Type: str8

The str8 type encodes up to 255 octets worth of UTF-8 unicode. The number of octets of unicode is first encoded as an 8-bit unsigned integral value. This is followed by the actual UTF-8 unicode. Note that the encoded size refers to the number of octets of unicode, not necessarily the number of characters since the UTF-8 unicode may include multi-byte character sequences.

Wire Format

```
  1 OCTET   size OCTETs
+---------+--------------+
|  size   | utf8-unicode |
+---------+--------------+
   uint8
```

6.10.4 Type: str8-utf16

The str8-utf16 type encodes up to 255 octets worth of UTF-16 unicode. The number of octets of unicode is first encoded as an 8-bit unsigned integral value. This is followed by the actual UTF-16 unicode. Note that the encoded size refers to the number of octets of unicode, not the number of characters since the UTF-16 unicode will include at least two octets per unicode character.

Wire Format

```
  1 OCTET    size OCTETs
+---------+---------------+
|  size   | utf16-unicode |
+---------+---------------+
   uint8
```

6.11 Variable Width (2 octet size)

6.11.1 Type: vbin16

The vbin16 type encodes up to 65535 octets of opaque binary data. The number of octets is first encoded as a 16-bit unsigned integral value in network byte order. This is followed by the actual data.

Wire Format

```
  2 OCTETs   size OCTETs
+----------+-------------+
|   size   |   octets    |
+----------+-------------+
   uint16
```

6.11.2 Type: str16-latin

The str16-latin type encodes up to 65535 octets of iso-8859-15 characters. The number of octets is first encoded as a 16-bit unsigned integral value in network byte order. This is followed by the actual characters.

Wire Format

```
  2 OCTETs        size OCTETs
+----------+------------------------+
|   size   |       characters       |
+----------+------------------------+
   uint16    iso-8859-15 characters
```

6.11.3 Type: str16

The str16 type encodes up to 65535 octets worth of UTF-8 unicode. The number of octets is first encoded as a 16-bit unsigned integral value in network byte order. This is followed by the actual UTF-8 unicode. Note that the encoded size refers to the number of octets of unicode, not necessarily the number of unicode characters since the UTF-8 unicode may include multi-byte character sequences.

Wire Format

```
  2 OCTETs   size OCTETs
+----------+--------------+
|   size   | utf8-unicode |
+----------+--------------+
   uint16
```

6.11.4 Type: str16-utf16

The str16-utf16 type encodes up to 65535 octets worth of UTF-16 unicode. The number of octets is first encoded as a 16-bit unsigned integral value in network byte order. This is followed by the actual UTF-16 unicode. Note that the encoded size refers to the number of octets of unicode, not the number of unicode characters since the UTF-16 unicode will include at least two octets per unicode character.

Wire Format

```
  2 OCTETs    size OCTETs
+----------+---------------+
|   size   | utf16-unicode |
+----------+---------------+
   uint16
```

6.12 Variable Width (4 octet size)

6.12.1 Type: vbin32

The vbin32 type encodes up to 4294967295 octets of opaque binary data. The number of octets is first encoded as a 32-bit unsigned integral value in network byte order. This is followed by the actual data.

Wire Format

```
  4 OCTETs   size OCTETs
+----------+-------------+
|   size   |   octets    |
+----------+-------------+
   uint32
```

6.12.2 Type: map

A map is a set of distinct keys where each key has an associated (type,value) pair. The triple of the key, type, and value, form an entry within a map. Each entry within a given map MUST have a distinct key. A map is encoded as a size in octets, a count of the number of entries, followed by the encoded entries themselves.

An encoded map may contain up to (4294967295 - 4) octets worth of encoded entries. The size is encoded as a 32-bit unsigned integral value in network byte order equal to the number of octets worth of encoded entries plus 4. (The extra 4 octets is added for the entry count.) The size is then followed by the number of entries encoded as a 32-bit unsigned integral value in network byte order. Finally the entries are encoded sequentially.

An entry is encoded as the key, followed by the type, and then the value. The key is always a string encoded as a str8. The type is a single octet that may contain any valid AMQP type code. The value is encoded according to the rules defined by the type code for that entry.

Wire Format

```
           +------------= size OCTETs =-----------+
           |                                      |
  4 OCTETs | 4 OCTETs                             |
+----------+----------+-----+---------------+-----+
|   size   |  count   | .../|     entry     |\... |
+----------+----------+---/ +---------------+ \---+
   uint32     uint32     / /                 \ \
                        / /                   \ \
                       / /                     \ \
                      / /                       \ \
                     / /                         \ \
                    / k OCTETs   1 OCTET   n OCTETs \
                   +-----------+---------+-----------+
                   |    key    |  type   |   value   |
                   +-----------+---------+-----------+
                       str8                 *type*
```

6.12.3 Type: list

A list is an ordered sequence of (type, value) pairs. The (type, value) pair forms an item within the list. The list may contain items of many distinct types. A list is encoded as a size in octets, followed by a count of the number of items, followed by the items themselves encoded in their defined order.

An encoded list may contain up to (4294967295 - 4) octets worth of encoded items. The size is encoded as a 32-bit unsigned integral value in network byte order equal to the number of octets worth of encoded items plus 4. (The extra 4 octets is added for the item count.) The size is then followed by the number of items encoded as a 32-bit unsigned integral value in network byte order. Finally the items are encoded sequentially in their defined order.

An item is encoded as the type followed by the value. The type is a single octet that may contain any valid AMQP type code. The value is encoded according to the rules defined by the type code for that item.

Wire Format

```
           +---------= size OCTETs =---------+
           |                                 |
  4 OCTETs | 4 OCTETs                        |
+----------+----------+-----+----------+-----+
|   size   |  count   | .../|   item   |\... |
+----------+----------+---/ +----------+ \---+
   uint32     uint32     / /            \ \
                        / /              \ \
                       / 1 OCTET   n OCTETs \
                      +----------+-----------+
                      |   type   |   value   |
                      +----------+-----------+
                                    *type*
```

6.12.4 Type: array

An array is an ordered sequence of values of the same type. The array is encoded in as a size in octets, followed by a type code, then a count of the number values in the array, and finally the values encoded in their defined order.

An encoded array may contain up to (4294967295 - 5) octets worth of encoded values. The size is encoded as a 32-bit unsigned integral value in network byte order equal to the number of octets worth of encoded values plus 5. (The extra 5 octets consist of 4 octets for the count of the number of values, and one octet to hold the type code for the items in the array.) The size is then followed by a single octet that may contain any valid AMQP type code. The type code is then followed by the number of values encoded as a 32-bit unsigned integral value in network byte order. Finally the values are encoded sequentially in their defined order according to the rules defined by the type code for the array.

Wire Format

```
  4 OCTETs   4 OCTETs   1 OCTET      (size - 5) OCTETs
+----------+----------+---------+-------------------------+
|   size   |  count   |  type   |         values          |
+----------+----------+---------+-------------------------+
   uint32     uint32              *count* encoded *type*s
```

6.12.5 Type: struct

The struct type provides a polymorphic encoding of any struct with a defined struct code. A struct is encoded as a size in octets followed by the struct code, followed by the encoded struct data. The struct code uniquely identifies the struct definition that determines the encoding of the struct data. The size is encoded as a 32-bit unsigned integral value in network byte order that is equal to the size of the encoded struct data plus the 2 OCTET struct code. See derived-types for a complete definition of encoded struct data.

Wire Format

```
  4 OCTETs   2 OCTETs         n OCTETs
+----------+----------+----------------------+
|   size   |   code   |  encoded-struct-data |
+----------+----------+----------------------+
   uint32

                n = (size - 2)
```

6.13 Fixed Width (5 octets)

6.13.1 Type: bin40

The bin40 type consists of five consecutive octets of opaque binary data.

Wire Format

```
  1 OCTET     1 OCTET      1 OCTET      1 OCTET      1 OCTET
+-----------+-----------+-------------+------------+------------+
| octet-one | octet-two | octet-three | octet-four | octet-five |
+-----------+-----------+-------------+------------+------------+
```

6.13.2 Type: dec32

The dec32 type is decimal value with a variable number of digits following the decimal point. It is encoded as an 8-bit unsigned integral value representing the number of decimal places. This is followed by the signed integral value encoded using a 32-bit two's complement representation in network byte order.

The former value is referred to as the exponent of the divisor. The latter value is the mantissa. The decimal value is given by: mantissa / 10^exponent.

Wire Format

```
  1 OCTET    4 OCTETs
+----------+----------+
| exponent | mantissa |
+----------+----------+
   uint8      int32
```

6.14 Fixed Width (9 octets)

6.14.1 Type: bin72

The bin72 type consists of nine consecutive octets of opaque binary data.

Wire Format

```
  1 OCTET      1 OCTET             1 OCTET       1 OCTET
+-----------+-----------+-----+-------------+------------+
| octet-one | octet-two | ... | octet-eight | octet-nine |
+-----------+-----------+-----+-------------+------------+
```

6.14.2 Type: dec64

The dec64 type is decimal value with a variable number of digits following the decimal point. It is encoded as an 8-bit unsigned integral value representing the number of decimal places. This is followed by the signed integral value encoded using a 64-bit two's complement representation in network byte order.

The former value is referred to as the exponent of the divisor. The latter value is the mantissa. The decimal value is given by: mantissa / 10^exponent.

Wire Format

```
  1 OCTET    8 OCTETs
+----------+----------+
| exponent | mantissa |
+----------+----------+
   uint8      int64
```

6.15 Zero Width

6.15.1 Type: void

The void type is used within tagged data structures such as maps and lists to indicate an empty value. The void type has no value and is encoded as an empty sequence of octets.

6.15.2 Type: bit

The bit type is used to indicate that a packing flag within a packed struct is being used to represent a boolean value based on the presence of an empty value. The bit type has no value and is encoded as an empty sequence of octets.

6.15.3 Type: any

The any type provides a polymorphic encoding of any coded value. It consists of a type code followed by the encoded value. The any type does not have a type code.

Wire Format

```
  1 OCTET
+---------+-------+
|  code   | value |
+---------+-------+
   uint8
```

7 Derived-types

7.1 Domains

An AMQP domain defines a new type with a format identical to another type, but with a restricted range of values. In some cases a closed set of permitted values is formally restricted to a predetermined set of values, and in other cases an open set of restricted values is specified.

7.2 Structs

An AMQP struct defines a compound type. That is a type whose format is defined entirely in terms of other types. Each struct definition includes an ordered sequence of well known fields, each with a specified name, type, and multiplicity. A struct is encoded as a set of packing flags that indicate which fields are present, followed by the encoded field data for those fields that are present. Where fields are defined to permit multiple values, the encoded data for that field starts with a 4 OCTET count indicating how many values are present.

The first and second octet of packing flags contain presence indicators for fields 1-8 and fields 9-16 respectively. Within each octet the fields map in order from the least significant bit to the most significant bit. If a packing flag is set the corresponding field MUST be included in the encoded data. If a packing flag is **not** set the corresponding field MUST NOT be included in the encoded data. If the struct has fewer properties than packing flags the extra packing flags are reserved for future extension of the struct and MUST be set to zero.

Structs are specifically defined by the command, control, and struct definitions within this document. Each definition includes a unique struct code for use with the struct primitive type as well as the order, name, type, and multiplicity of the struct fields.

Example

```
     only included if corresponding flags are set
                           |
                 +---------+---------+
                 |         |         |
   2 octets     \|/       \|/       \|/
 +----------+---------+---------+---------+------
 |  flags  /| field-1 |\field-2 |\field-3 |\ ...
 +--------/-+---------+-\-------+-\-------+-\----
         / /           \ \       \ \       \ \
        / /             \ \       \ \       \ \
       / /               \ \       \ \       \ \
      / /                 \ \       \ \       \ \
     / /                   \ \       \ \       \ \
    / /                     \ \       \ \       \ \
   / /     multiple=true     \ \       \ \       \ \
  / /                         \ \       \ \       \ \
 / /n                          \ \       \ \       \ \
+-------+---------+-----+---------+       \ +---------+
| count | value-1 | ... | value-n |        \|  value  |
+-------+---------+-----+---------+         +---------+
 4 OCTET
```

7.3 Records

An AMQP record defines a compound type similar to a struct. Unlike structs, records are encoded without packing flags, and each field must always be present on the wire.

Example

```
                    always present
                           |
                 +---------+---------+
                 |         |         |
                \|/       \|/       \|/
            +---------+---------+---------+------
           /| field-1 |\field-2 |\field-3 |\ ...
          / +---------+-\-------+-\-------+-\----
         / /           \ \       \ \       \ \
        / /             \ \       \ \       \ \
       / /               \ \       \ \       \ \
      / /                 \ \       \ \       \ \
     / /                   \ \       \ \       \ \
    / /                     \ \       \ \       \ \
   / /     multiple=true     \ \       \ \       \ \
  / /                         \ \       \ \       \ \
 / /n                          \ \       \ \       \ \
+-------+---------+-----+---------+       \ +---------+
| count | value-1 | ... | value-n |        \|  value  |
+-------+---------+-----+---------+         +---------+
 4 OCTET
```