# AMQP 1.0 DRAFT for Review Only

Revision: ***PR3***

Creation Date: 2010-05-05 20:35:15
Subversion Rev: 1017

# License

# Table of Contents

# 1 Introduction to AMQP

## 1.1 Overview

The Advanced Message Queuing Protocol (AMQP) is an open Internet Protocol for Business Messaging.

AMQP is divided up into separate layers. At the lowest level we define an efficient binary peer-to-peer protocol for transporting messages between two processes over a network. Secondly we define an abstract message format, with concrete standard encoding. Every compliant AMQP process must be able to send and receive messages in this standard encoding.

## 1.2 Rationale and Use Cases

A community of business messaging users defined the requirements for AMQP based on their experiences of building and operating networked information processing systems.

The AMQP Working Group measures the success of AMQP according to how well the protocol satisfies these requirements, as outlined below. The development of these requirements is not static the most recent list can be found at *http://www.amqp.org*.

| ***AMQP Release*** | ***User Requirement*** |
|---|---|
| | ***Ubiquity*** |
| 1.0 | Open Internet protocol standard supporting unencumbered (a) use, (b) implementation, and (c) extension |
| | Clear and unambiguous core functionality for business message routing and delivery within Internet infrastructure - so that business messaging is provided by infrastructure and not by integration experts |
| | Low barrier to understand, use and implement |
| 1.0 | Fits into existing enterprise messaging applications environments in a practical way |
| | ***Safety*** |
| 1.0 | Infrastructure for a secure and trusted global transaction network<br>• Consisting of business messages that are tamper proof<br>• Supporting message durability independent of receivers being connected, and<br>• Message delivery is resilient to technical failure |
| 1.0 | Supports business requirements to transport business transactions of any financial value |
| Future | Sender and Receiver are mutually agreed upon counter parties - No possibility for injection of Spam |
| | ***Fidelity*** |
| 1.0 | Well-stated message queueing and delivery semantics covering: at-most-once; at-least-once; and once-and-only-once aka 'reliable' |
| 1.0 | Well-stated message ordering semantics describing what a sender can expect (a) a receiver to observe and (b) a queue manager to observe |
| 1.0 | Well-stated reliable failure semantics so all exceptions can be managed |
| | ***Applicability*** |
| | As TCP subsumed all technical features of networking, we aspire for AMQP to be the prevalent business messaging technology (tool) for organizations so that with increased use, ROI increases |

| | |
|---|---|
| | and TCO decreases |
| 1.0 | Any AMQP client can initiate communication with, and then communicate with, any AMQP broker over TCP |
| Future | Any AMQP client can request communication with, and if supported negotiate the use of, alternate transport protocols (e.g. SCTP, UDP/Multicast), from any AMQP broker |
| 1.0 | Provides the core set of messaging patterns via a single manageable protocol: asynchronous directed messaging, request/reply, publish/subscribe, store and forward |
| 1.0 | Supports Hub & Spoke messaging topology within and across business boundaries |
| Future | Supports Hub to Hub message relay across business boundaries through enactment of explicit agreements between broker authorities |
| | Supports Peer to Peer messaging across any network |
| | ***Interoperability*** |
| 1.0 | Multiple stable and interoperating broker implementations each with a completely independent provenance including design, architecture, code, ownership |
| 1.0 | Each broker implementation is conformant with the specification, for all mandatory message exchange and queuing functionality, including fidelity semantics |
| 1.0 | Implementations are independently testable and verifiable by any member of the public free of charge |
| 1.0 | Stable core (client-broker) wire protocol so that brokers do not require upgrade during 1.x feature evolution: Any 1.x client will work with any 1.y broker if y >= x |
| Future | Stable extended (broker-broker) wire protocol so that brokers do not require upgrade during 1.x feature evolution: Any two brokers versions 1.x, 1.y can communicate using protocol 1.x if x<y |
| 1.0 | Layered architecture, so features & network transports can be independently extended by separated communities of use, enabling business integration with other systems without coordination with the AMQP Working Group |
| | ***Manageability*** |
| 1.0 | Binary WIRE protocol so that it can be ubiquitous, fast, embedded (XML can be layered on top), enabling management to be provided by encapsulating systems (e.g. O/S, middleware, phone) |
| 1.0 | Scalable, so that it can be a basis for high performance fault-tolerant lossless messaging infrastructure, i.e. without requiring other messaging technology |
| Future | Interaction with the message delivery system is possible, sufficient to integrate with prevailing business operations that administer messaging systems using management standards. |
| Future | Intermediated: supports routing and relay management, traffic flow management and quality of service management |
| Future | Decentralized deployment with independent local governance |
| Future | Global addressing standardizing end to end delivery across any network scope |

## 1.3 How to Read the Standard

The AMQP standard is divided into Books which define the separate parts of the standard. Depending on your area of interest you may wish to start reading a particular Book and use the other Books for reference.

| | |
|---|---|
| Book I | Explains the messaging model that underpins AMQP |
| Book II | Explains the AMQP Type System used both by the Messaging and Transport |

|  |  |
|---|---|
|  | Layers |
| Book III | Defines the AMQP Transport Layer |
| Book IV | Defines the AMQP Messaging Layer |
| Book V | Defines the AMQP Security Layers |

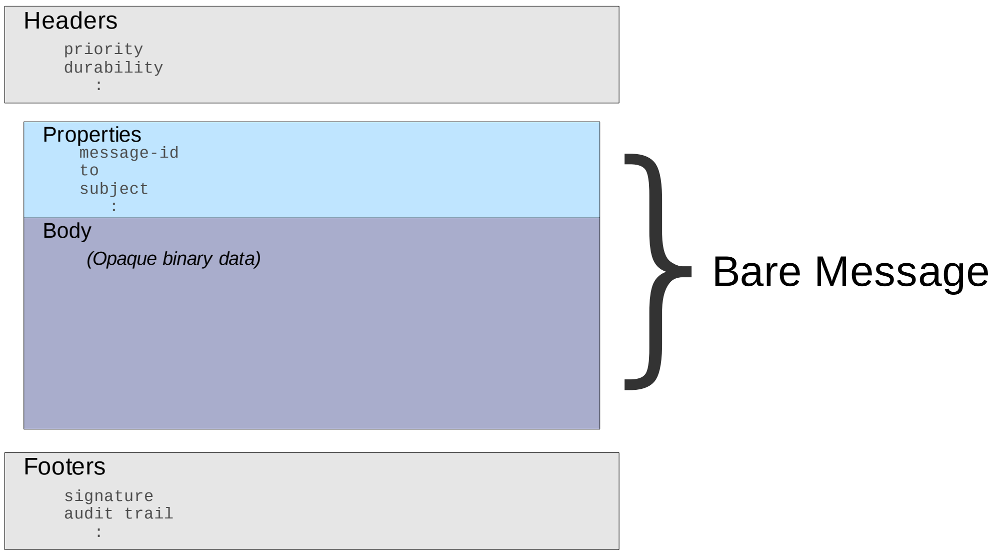
# Book I - Messaging Model

# 1 Messages

The term message is used with various connotations in the messaging world. The sender may like to think of the message as an immutable payload handed off to the messaging infrastructure for delivery. The receiver often thinks of the message as not only that immutable payload from the sender, but also various annotations supplied by the messaging infrastructure along the way. To avoid confusion we formally define the term **bare message** to mean the message as supplied by the sender and the term **annotated message** to mean the message as seen at the receiver. The term **message** itself should only be used where the distinction is irrelevant.

An **annotated message** consists of the **bare message** plus an area for annotations. There are two classes of annotations. Annotations that travel with the message indefinitely, and annotations that are consumed by the next node.

A **bare message** consists of a uniquely identified payload of application data divided into readable “properties” and an opaque “body”.



The bare message is immutable within the AMQP network.  That is neither the opaque body, nor the properties can be changed by any node without it becoming a different message (and thus requiring a new identifier).

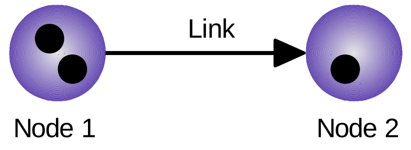
Information which may be modified by the network, or information which is directed at the infrastructure and does not form part of the bare message, is placed in a “header” and “footer”. The header and footer are transferred with the message but may be updated en-route from its origin to its destination.

# 2 Nodes and Links

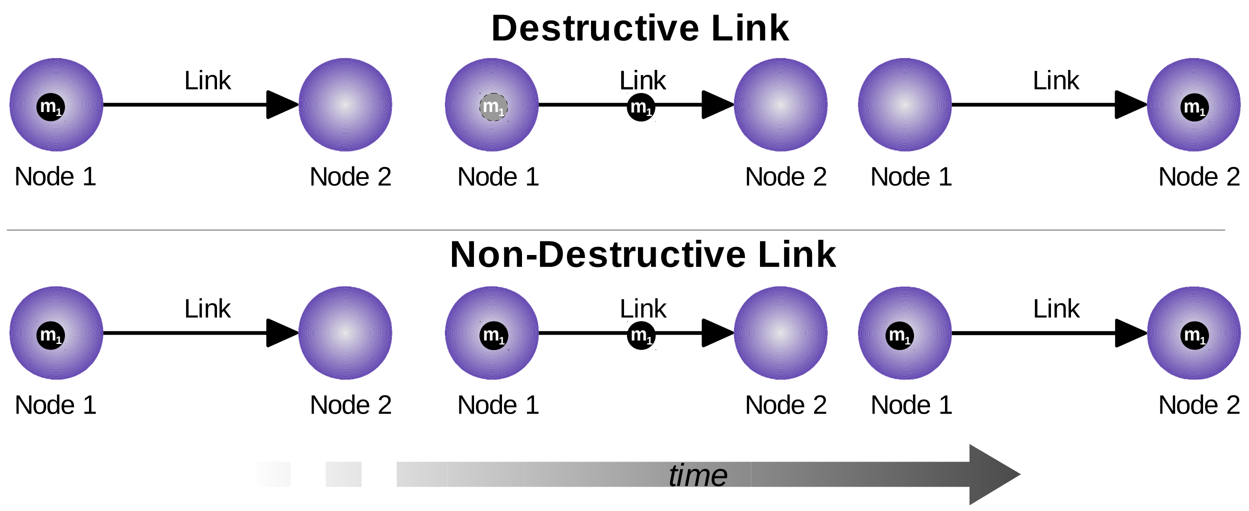
An AMQP Network consists of **Nodes** and **Links**.

A Node is a named source and/or sink of **Messages** (see Chapter 1). A Message is created at a (Producer) Node, and may travel along links, via other nodes until it reaches a terminating (Consumer) Node.

A Link is a unidirectional route between nodes along which messages may travel. Links may have entry criteria (**Filters**) which restrict which messages may travel along them. The link lifetime is tied to the lifetime of the source and destination nodes. If the node at either end of the link is deleted, so is the link itself.

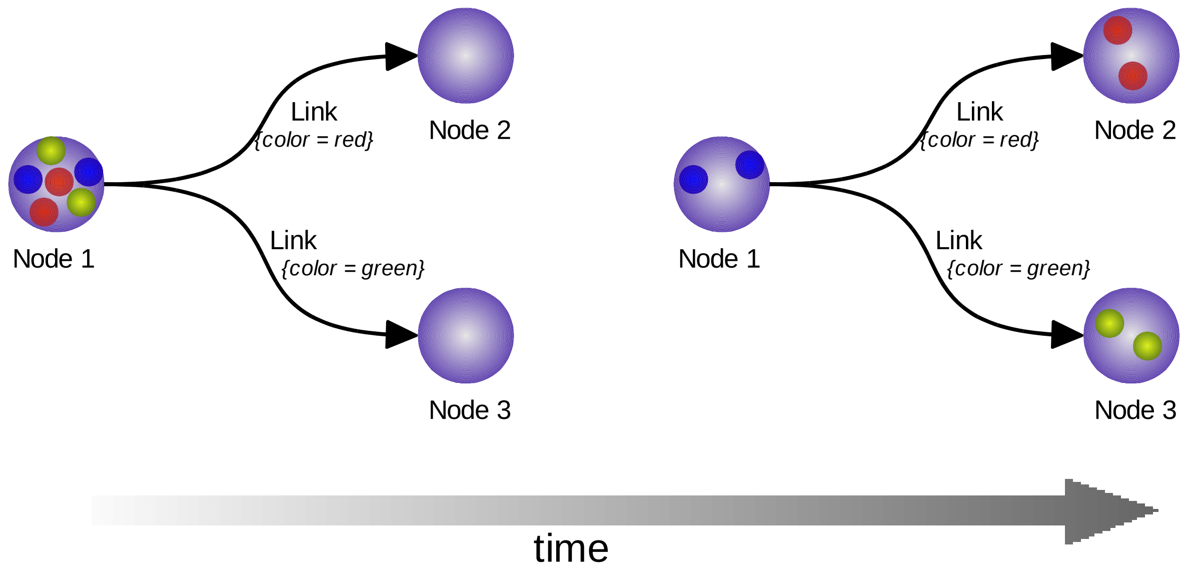


Links may be "destructive", or "non-destructive". When a message is sent along a destructive link then after the transfer to the destination node has completed, the message is no longer present at the source node. For a non-destructive link the transfer is closer to a "copy" since on completion of the transfer of the message the original still exists in the source node.



In the Non-Destructive Link example the message $m_1$ is now simultaneously present at more than one node. More generally in AMQP, a single message identified by its id, can be simultaneously present in many different nodes, however a message can be present at most once at any given node. That is, if a message *M* arrives at a node *N* where *M* is already present, then there is no change to the set of messages at *N*. More generally nodes should treat as a duplicate the arrival of any message *M′* which has the same identity as a message *M* which has previously arrived at the node. Duplicate deliveries should be discarded.

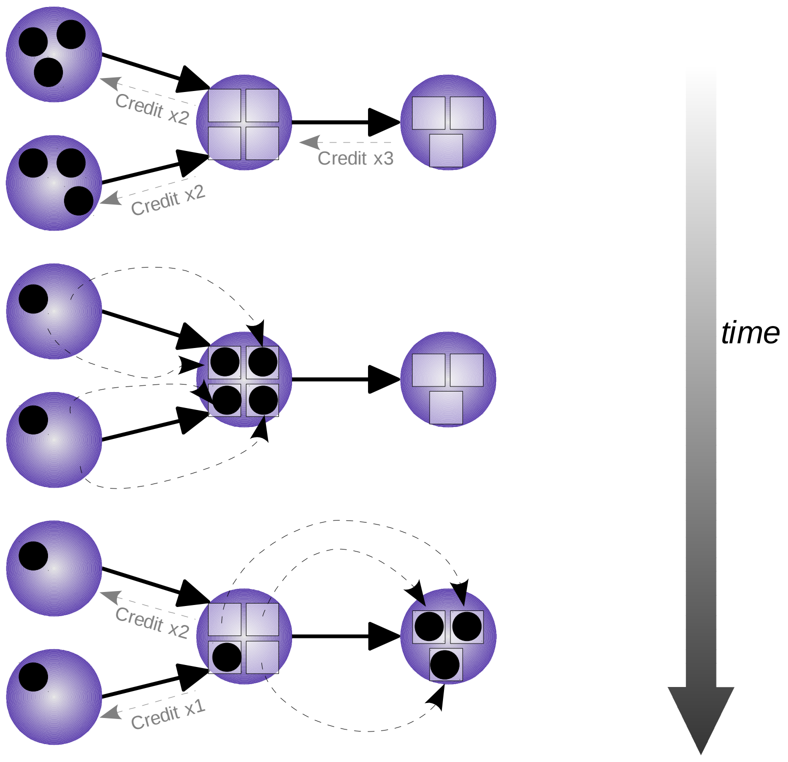
By using filters on the outgoing links from a node, messages may be distributed based on the non-opaque portions of the message.



# 3 Credit

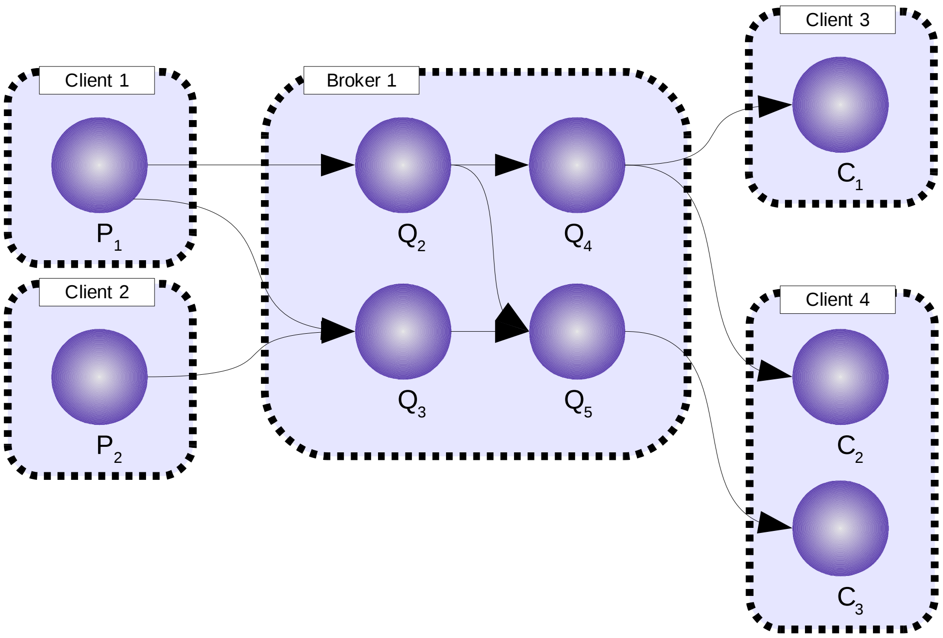
A message may only pass along a given link if the destination node has issued **credit** to the link. A link with no credit is impassable. The unit of credit is the “transfer unit”. Each message can be described in terms of the number of transfer units that are required to send it. While large messages may be several transfer units in size, most messages on any given system should require only one transfer unit of credit to send. Sending a message irrevocably consumes the credit. Receivers of messages can add or remove multiple units of credit at any time.

One can think of AMQP as a network where messages travel in one direction and an equal (or greater) amount of credit flows in the opposite direction. Credit is used to ensure that nodes do not receive more messages than they can physically store. If we think of each node having a given amount of free capacity at any one instant then we can see that an amount of credit equal to the free-space at that node can be distributed amongst its incoming links.



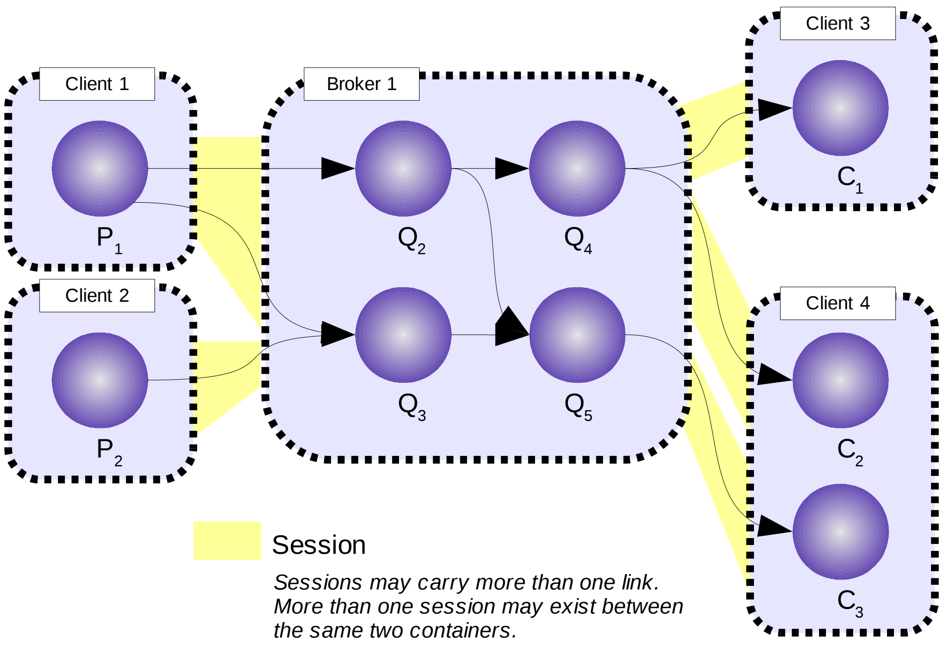
# 4 Containers

Nodes within the AMQP network exist within **Containers**. A container is a physical or logical process to which network connections can be established. Node names are unambiguous within a container (that is no two nodes will simultaneously have the same name within the container, though many names may relate to the same node). Containers have a globally unique identifier. The combination of container identifier and node name is thus a globally unambiguous identifier for any node. While container identifiers are unique they may not necessarily have any meaning outside of the context of an AMQP network.

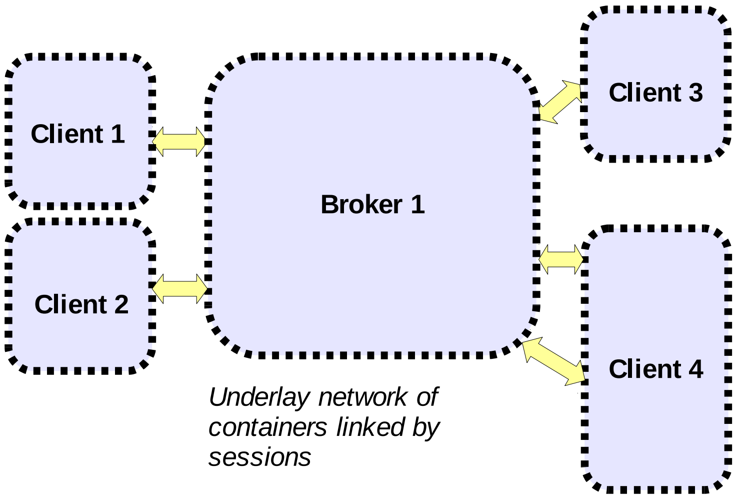


# 5 Sessions

A **Session** is a named full-duplex  interaction between two containers.  Links between nodes in different containers are created on a session.  Establishing a session requires (mutual) authentication.  There may be more than one session established between any two containers at any one time.  Each session may contain multiple links.


Client 1
Broker 1
Client 3
Client 2
Client 4
$P_1$
$P_2$
$Q_2$
$Q_3$
$Q_4$
$Q_5$
$C_1$
$C_2$
$C_3$
Session
*Sessions may carry more than one link.*
*More than one session may exist between the same two containers.*

Containers and Sessions form an underlay network, nodes and links an overlay network atop them.



# Book II - Data Types

# 1 Type System

The AMQP type system defines a set of commonly used primitive types used for interoperable data representation. AMQP values may be annotated with additional semantic information beyond that associated with the primitive type. This allows for the association of an AMQP value with an external type that is not present as an AMQP primitive. For example, a URL is commonly represented as a string, however not all strings are valid URLs, and many programming languages and/or applications define a specific type to represent URLs. The AMQP type system would allow for the definition of a code with which to annotate strings when the value is intended to represent a URL.

## 1.1 Primitive Types

The following primitive types are defined:

| | |
|---|---|
| null: | Indicates an empty value. |
| boolean: | Represents a true or false value. |
| ubyte: | Integer in the range 0 to 2^8 - 1. |
| ushort: | Integer in the range 0 to 2^16 - 1. |
| uint: | Integer in the range 0 to 2^32 - 1. |
| ulong: | Integer in the range 0 to 2^64 - 1. |
| byte: | Integer in the range -(2^7) to 2^7 - 1. |
| short: | Integer in the range -(2^15) to 2^15 - 1. |
| int: | Integer in the range -(2^31) to 2^31 - 1. |
| long: | Integer in the range -(2^63) to 2^63 - 1. |
| float: | 32-bit floating point number (IEEE 754-2008 binary32). |
| double: | 64-bit floating point number (IEEE 754-2008 binary64). |
| char: | A single unicode character. |
| timestamp: | An absolute point in time. |
| uuid: | A universally unique id as defined by RFC-4122 section 4.1.2. |
| binary: | A sequence of octets. |
| string: | A sequence of unicode characters. |
| symbol: | Symbols are values from a constrained domain. Although the set of possible domains is open-ended, typically the both number and size of symbols in use for any given application will be small, e.g. small enough that it is reasonable to cache all the distinct values. |
| list: | A sequence of polymorphic values. |
| map: | A polymorphic mapping from distinct keys to values. |

## 1.2 Decoding Primitive Types

For any given programming language there may not be a direct equivalence between the available native language types and the AMQP types. The following table provides recommendations as to how to decode AMQP typed values in a selection of popular languages.

### Special Values

| AMQP | Python | Java | C99 | C++ | Ruby | C# |
|---|---|---|---|---|---|---|
| null | None | null | NULL | 0 | nil | null |
| true | True | true | true | true | true | true |
| false | False | false | false | false | false | false |

### Suggested Mappings

| AMQP | Python | Java | C99 | C++ | Ruby | C# |
|---|---|---|---|---|---|---|
| null | NoneType | -- | -- | -- | NilClass | -- |
| boolean | bool | Boolean | bool | bool | TrueClass\|FalseClass | bool |
| ubyte | int | Short | uint8_t | uint8_t | | |
| ushort | int | Integer | uint16_t | uint16_t | | |
| uint | int\|long | Long | uint32_t | uint32_t | | |
| ulong | long | BigInteger | uint64_t | uint64_t | | |
| byte | int | Byte | int8_t | int8_t | | |
| short | int | Short | int16_t | int16_t | | |
| int | int | Integer | int32_t | int32_t | | |
| long | long | Long | int64_t | int64_t | | |
| float | float | Float | float | float | | |
| double | float | Double | double | double | | |
| char | str | Character | wchar | wchar | | |
| timestamp | datetime | Date | int64_t | int64_t | | |
| uuid | UUID | UUID | char[16] | char[16] | | |
| binary | str | byte[] | char[] | char[] | | |
| string | unicode | String | wchar[] | std::wstring | | |
| symbol | str | | | | | |
| list | list | List | | | | |
| map | dict | Map | | | | |

## 1.3 Described Types

The primitive types defined by AMQP can directly represent many of the basic types present in most popular programming languages, and therefore may be trivially used to exchange basic data. In practice, however, even the simplest applications have their own set of custom types used to model concepts within the application's domain, and, for messaging applications, these custom types need to be externalized for transmission.

AMQP provides a means to do this by allowing any AMQP type to be annotated with a *descriptor*. A *descriptor* forms an association between a custom type, and an AMQP type. This association indicates that the AMQP type is actually a *representation* of the custom type. The resulting combination of the AMQP type and its descriptor is referred to as a *described type*.

A described type contains two distinct kinds of type information. It identifies both an AMQP type and a custom type (as well as the relationship between them), and so can be understood at two different levels. An application with intimate knowledge of a given domain can understand described types as the custom types they represent, thereby decoding and processing them according to the complete semantics of the domain. An application with no intimate knowledge can still understand the described types as AMQP types, decoding and processing them as such.

## 1.4 Descriptor Values

Descriptor values other than symbolic (`symbol`) or numeric (`ulong`) are reserved, including numeric types other than `ulong`. The namespace for both symbolic and numeric descriptors is divided into distinct domains. Each domain has a defined symbol and/or 4 byte numeric id assigned by the AMQP working group. Descriptors are then assigned within each domain according to the following rules:

| | |
|---|---|
| **symbolic descriptors:** | *<domain>*:*<name>* |
| **numeric descriptors:** | (*domain-id* << 32) \| *descriptor-id* |

# 2 Type Encodings

An AMQP encoded data stream consists of untyped bytes with embedded constructors. The embedded constructor indicates how to interpret the untyped bytes that follow. Constructors can be thought of as functions that consume untyped bytes from an open ended byte stream and construct a typed value. An AMQP encoded data stream always begins with a constructor.

### Primitive Format Code (String)

```
       constructor            untyped bytes
            |                      |
          +--+   +-----------------+-----------------+
          |  |   |                                   |
     ...  0xA1   0x1E "Hello Glorious Messaging World"  ...
           |     |  |              |                 |
           |     |  |         utf8 bytes             |
           |     |  |                                |
           |     | # of data octets                  |
           |     |                                   |
           |     +-----------------+-----------------+
           |                       |
           |        string value encoded according
           |          to the str8-utf8 encoding
           |
  primitive format code
for the str8-utf8 encoding
```

An AMQP constructor consists of either a primitive format code, or a described format code. A primitive format code is a constructor for an AMQP primitive type. A described format code consists of a descriptor and a primitive format-code. A descriptor defines how to produce a domain specific type from an AMQP primitive value.

### Described Format Code (URL)

```
            constructor                       untyped bytes
                 |                                 |
     +-----------+-----------+   +-----------------+-----------------+
     |                       |   |                                   |
...  0x00 0xA1 0x03 "URL" 0xA1   0x1E "http://example.org/hello-world"  ...
          |             |  |     |                                   |
          +------+------+  |     |                                   |
                 |         |     |                                   |
            descriptor     |     +------------------+----------------+
                           |                        |
                           |         string value encoded according
                           |           to the str8-utf8 encoding
                           |
                  primitive format code
                for the str8-utf8 encoding
```

The descriptor portion of a described format code is itself any valid AMQP encoded value, including other described values. The formal BNF for constructors is given below.

### Constructor BNF

```
      constructor = format-code
                  / %x00 descriptor format-code

      format-code = fixed / variable / compound / array
            fixed = empty / fixed-one / fixed-two / fixed-four
                  / fixed-eight / fixed-sixteen
         variable = variable-one / variable-four
         compound = compound-one / compound-four
            array = array-one / array-four

       descriptor = value
            value = constructor untyped-bytes
    untyped-bytes = *OCTET ; this is not actually *OCTET, the
                           ; valid byte sequences are restricted
                           ; by the constructor

    ; fixed width format codes
            empty = %x40-4E / %x4F %x00-FF
        fixed-one = %x50-5E / %x5F %x00-FF
        fixed-two = %x60-6E / %x6F %x00-FF
       fixed-four = %x70-7E / %x7F %x00-FF
      fixed-eight = %x80-8E / %x8F %x00-FF
    fixed-sixteen = %x90-9E / %x9F %x00-FF

    ; variable width format codes
     variable-one = %xA0-AE / %xAF %x00-FF
    variable-four = %xB0-BE / %xBF %x00-FF

    ; compound format codes
     compound-one = %xC0-CE / %xCF %x00-FF
    compound-four = %xD0-DE / %xDF %x00-FF

    ; array format codes
        array-one = %xE0-EE / %xEF %x00-FF
       array-four = %xF0-FE / %xFF %x00-FF
```

Format codes map to one of four different categories: fixed width, variable width, compound and array. Values encoded within each category share the same basic structure parameterized by width. The subcategory within a format-code identifies both the category and width.

**Fixed Width:** The size of fixed-width data is determined based solely on the subcategory of the format code for the fixed width value.

**Variable Width:** The size of variable-width data is determined based on an encoded size that prefixes the data. The width of the encoded size is determined by the subcategory of the format code for the variable width value.

**Compound:** Compound data is encoded as a size and a count followed by a polymorphic sequence of *count* constituent values. Each constituent value is preceded by a constructor that indicates the semantics and encoding of the data that follows. The width of the size and count is determined by the subcategory of the format code for the compound value.

**Array:** Array data is encoded as a size and count followed by an array element constructor followed by a monomorphic sequence of values encoded according to the supplied array element constructor. The width of the size and count is determined by the subcategory of the format code for the array.

The bits within a format code may be interpreted according to the following layout:

```
Bit:  7    6    5    4    3    2    1    0
     +------------------------------------+ +----------+
     |   subcategory   |      subtype     | | ext-type |
     +------------------------------------+ +----------+
                   1 octet                   1 octet
     |                                                 |
     +-------------------------------------------------+
                              |
                         format-code

         ext-type: only present if subtype is 0xF
```

The following table describes the subcategories of format-codes:

| Subcategory | Category | Format |
|---|---|---|
| 0x4 | Fixed Width | Zero octets of data. |
| 0x5 | Fixed Width | One octet of data. |
| 0x6 | Fixed Width | Two octets of data. |
| 0x7 | Fixed Width | Four octets of data. |
| 0x8 | Fixed Width | Eight octets of data. |
| 0x9 | Fixed Width | Sixteen octets of data. |
| 0xA | Variable Width | One octet of size, 0-255 octets of data. |
| 0xB | Variable Width | Four octets of size, 0-4294967295 octets of data. |
| 0xC | Compound | One octet each of size and count, 0-255 distinctly typed values. |
| 0xD | Compound | Four octets each of size and count, 0-4294967295 distinctly typed values. |
| 0xE | Array | One octet each of size and count, 0-255 uniformly typed values. |
| 0xF | Array | Four octets each of size and count, 0-4294967295 uniformly typed values. |

Please note, unless otherwise specified, AMQP uses network byte order for all numeric values.

## 2.1 Fixed Width

The width of a specific fixed width encoding may be computed from the subcategory of the format code for the fixed width value:

```
  n OCTETs
+----------+
|   data   |
+----------+
```

| Subcategory | n |
|---|---|
| 0x4 | 0 |
| 0x5 | 1 |
| 0x6 | 2 |
| 0x7 | 4 |
| 0x8 | 8 |
| 0x9 | 16 |

## 2.2 Variable Width

All variable width encodings consist of a size in octets followed by *size* octets of encoded data.

The width of the size for a specific variable width encoding may be computed from the subcategory of the format code:

```
  n OCTETs   size OCTETs
+----------+-------------+
|   size   |    value    |
+----------+-------------+

    Subcategory     n
    =================
    0xA             1
    0xB             4
```

## 2.3 Compound

All compound encodings consist of a size and a count followed by *count* encoded items. The width of the size and count for a specific compound encoding may be computed from the category of the format code:

```
                      +----------= count items =-----------+
                      |                                    |
  n OCTETs   n OCTETs |                                    |
+----------+----------+--------------+------------+--------+
|   size   |  count   |      ...    /|    item    |\ ...   |
+----------+----------+------------/ +------------+ \------+
                                  / /               \ \
                                 / /                 \ \
                                / /                   \ \
                               +-------------+-----------+
                               | constructor |   data    |
                               +-------------+-----------+

                   Subcategory     n
                   =================
                   0xC             1
                   0xD             4
```

## 2.4 Array

All array encodings consist of a size followed by a count followed by an element constructor followed by *count* elements of encoded data formated as required by the element constructor:

```
                                                  +--= count elements ---+
                                                  |                      |
  n OCTETs   n OCTETs                             |                      |
+----------+----------+---------------------+-------+------+-------+
|   size   |  count   | element-constructor |  ...  | data |  ...  |
+----------+----------+---------------------+-------+------+-------+

                        Subcategory     n
                        =================
                        0xE             1
                        0xF             4
```

## 2.5 List of Encodings

| Type | Encoding | Code | Category/Width | Description |
|---|---|---|---|---|
| null | | 0x40 | fixed/0 | The null value. |
| boolean | true | 0x41 | fixed/0 | The boolean value true. |
| boolean | false | 0x42 | fixed/0 | The boolean value false. |
| ubyte | | 0x50 | fixed/1 | 8-bit unsigned integer. |
| ushort | | 0x60 | fixed/2 | 16-bit unsigned integer in network byte order. |
| uint | | 0x70 | fixed/4 | 32-bit unsigned integer in network byte order. |
| ulong | | 0x80 | fixed/8 | 64-bit unsigned integer in network byte order. |
| byte | | 0x51 | fixed/1 | 8-bit two's-complement integer. |
| short | | 0x61 | fixed/2 | 16-bit two's-complement integer in network byte order. |
| int | | 0x71 | fixed/4 | 32-bit two's-complement integer in network byte order. |
| long | | 0x81 | fixed/8 | 64-bit two's-complement integer in network byte order. |
| float | ieee-754 | 0x72 | fixed/4 | IEEE 754-2008 binary32. |
| double | ieee-754 | 0x82 | fixed/8 | IEEE 754-2008 binary64. |
| char | utf32 | 0x73 | fixed/4 | A UTF-32 encoded unicode character. |
| timestamp | ms64 | 0x83 | fixed/8 | Encodes a point in time using a 64 bit signed integer representing milliseconds since Midnight Jan 1, 1970 UTC. For the purpose of this representation, milliseconds are taken to be (1/(24*60*60*1000))th of a day. |
| uuid | | 0x98 | fixed/16 | UUID as defined in section 4.1.2 of RFC-4122. |
| binary | vbin8 | 0xa0 | variable/1 | Up to 2^8 - 1 octets of binary data. |
| binary | vbin32 | 0xb0 | variable/4 | Up to 2^32 - 1 octets of binary data. |
| string | str8-utf8 | 0xa1 | variable/1 | Up to 2^8 - 1 octets worth of UTF-8 unicode. |
| string | str8-utf16 | 0xa2 | variable/1 | Up to 2^8 - 1 octets worth of UTF-16 unicode. |
| string | str32-utf8 | 0xb1 | variable/4 | Up to 2^32 - 1 octets worth of UTF-8 unicode. |
| string | str32-utf16 | 0xb2 | variable/4 | Up to 2^32 - 1 octets worth of UTF-16 unicode. |
| symbol | sym8 | 0xa3 | variable/1 | Up to 2^8 - 1 seven bit ASCII characters representing a symbolic value. |
| symbol | sym32 | 0xb3 | variable/4 | Up to 2^32 - 1 seven bit ASCII characters representing a symbolic value. |
| list | list8 | 0xc0 | compound/1 | Up to 2^8 - 1 list elements with total size less than 2^8 octets. |
| list | list32 | 0xd0 | compound/4 | Up to 2^32 - 1 list elements with total size less than 2^32 octets. |
| list | array8 | 0xe0 | array/1 | Up to 2^8 - 1 array elements with total size less than 2^8 octets. |
| list | array32 | 0xf0 | array/4 | Up to 2^32 - 1 array elements with total size less than 2^32 octets. |
| map | map8 | 0xc1 | compound/1 | Up to 2^8 - 1 octets of encoded map data.<br>A map is encoded as a compound value where the constituent elements form alternating key value pairs.<br>`  item 0   item 1      item n-1    item n`<br>`+-------+-------+----+---------+---------+`<br>`\| key 1 \| val 1 \| .. \| key n/2 \| val n/2 \|`<br>`+-------+-------+----+---------+---------+` |
| map | map32 | 0xd1 | compound/4 | Up to 2^32 - 1 octets of encoded map data. See map8 above for a definition |

| | | | |
|---|---|---|---|
| | | | of encoded map data. |

# 3 Compound Types

AMQP defines a number of *compound types* used for encoding structured data such as commands, and controls. A compound type describes a composite value where each constituent value is identified by a well known named *field*. Each compound type definition includes an ordered sequence of fields, each with a specified name, type, and multiplicity. Compound type definitions also include one or more descriptors (symbolic and/or numeric) for identifying their defined representations.

Compound types are formally defined in the XML documents included with the specification. The following notation is used to define them:

#### Example Compound Type

```
<type class="compound" name="book" label="example compound type">
  <doc>
    <p>An example compound type.</p>
  </doc>

  <descriptor name="example:book:list" code="0x00000003:0x00000002"/>

  <field name="title" type="string" required="true" label="title of the book"/>

  <field name="authors" type="string" multiple="true"/>

  <field name="isbn" type="string" label="the isbn code for the book"/>
</type>
```

The *required* attribute of a field description controls whether a null element value is permitted in the representation. The *multiple* attribute of a field description controls whether a list of values is permitted in the representation. A single element of the type specified in the field description is always permitted, and lists if used MUST use an element type that matches the field description.

AMQP compound types may be encoded either as a described list or a described map. In general a specific compound type definition will allow for only one encoding option. This is currently indicated by the naming convention of the symbolic descriptor. Map encodings have symbolic descriptors ending in ":map", and list encodings in ":list". (Note that there may be a more formal way of distinguishing in a future revision.)

## 3.1 List Encoding

When encoding AMQP compound values as a described list, each element in the list is positionally correlated with the fields listed in the compound type definition. The permitted element values are determined by the type specification and multiplicity of the corresponding field definitions. When the trailing elements of the list representation are null, they MAY be omitted. The compound-value descriptor of the list indicates the specific compound type being represented.

The described list shown below is an example compound value of the *book* type defined above. A trailing null element corresponding to the absence of an isbn value is depicted in the example, but may optionally be omitted according to the encoding rules.

### Example Compound Value

```
             constructor                  list representation of a book
                  |                                   |
+-----------------+-------------------+ +-------------+---------------+
|                                     | |                             |
0x00 0xA3 0x11 "example:book:list" 0xC0 0x40 0x03  title  authors  isbn
      |                 |                            |       |       |
      |      identifies compound type                |       |       |
      |                                              |       |     0x40
    sym8                      +----------------------+       |       |
  (symbol)                    |                              |  null value
               +--------------+----------------+             |
               |                               |             |
               0xA1 0x15 "AMQP for & by Dummies"             |
                                                             |
+------------------------------------------------------------+-----+
|                                                                  |
0xE0 0x25 0x02 0xA1 0x0E "Rob J. Godfrey" 0x13 "Rafael H. Schloming"
      |    |    |   |                   | |                        |
    size   |    |   +---------+---------+ +-----------+------------+
           |    |             |                       |
         count  |        first element          second element
                |
        element constructor
```

## 3.2 Map Encoding

When encoding AMQP compound values as a described map, the set of valid keys is restricted to symbols identifying the field names in the compound type definition. The value associated with a given key MUST match the type specification and multiplicity from the field definition. If there is no value associated with a given field, the key MAY be omitted from the map. This is the same as supplying a null value for the given key. The compound-value descriptor of the map indicates the specific compound type being represented.

# Book III - Transport

# 1 Transport

The AMQP Network consists of *Nodes* connected via *Links*. Nodes are named entities responsible for the safe storage and/or delivery of Messages. Messages can originate from, terminate at, or be relayed by Nodes. A Link is a unidirectional route between two Nodes along which Messages may travel if they meet the entry criteria of the Link. As a Message travels through the AMQP network, the responsibility for safe storage and delivery of the Message is transferred between the Nodes it encounters. The Link Protocol (defined in the links section) manages the transfer of responsibility between two Nodes.

```
  +------------+                           +------------+
 /    Node A    \                         /    Node B    \
+----------------+   +--filter           +----------------+
|                |   |                   |                |
|  MSG_3 <MSG_1> |  +-+                  |         MSG_1  |
|                |--|?|----------------->|                |
| <MSG_2> MSG_4  |  +-+                  |  MSG_2         |
|                |       Link(A,B)       |                |
+----------------+                       +----------------+

           Key: <MSG_n> = old location of MSG_n
```

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.

```
   +---------------+                     +----------+
   | <<Container>> | 1..1           0..n | <<Node>> |
   |---------------|<>------------------>|----------|
   | container-id  |                     | name     |
   +---------------+                     +----------+
          /_\                                  /_\
           |                                    |
           |                                    |
     +-----+-----+                     +----------+----------+
     |           |                     |          |          |
     |           |                     |          |          |
+--------+  +--------+      +----------+  +----------+  +-------+
| Broker |  | Client |      | Producer |  | Consumer |  | Queue |
|--------|  |--------|      |----------|  |----------|  |-------|
|        |  |        |      |          |  |          |  |       |
+--------+  +--------+      +----------+  +----------+  +-------+
```

The AMQP Transport Specification 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.

Containers communicate via *Connections*. An AMQP Connection consists of a full-duplex, reliably ordered sequence of *Frames*. The precise requirement for a Connection is that if the n-th Frame arrives, all Frames prior to n MUST also have arrived. It is assumed Connections are transient and may fail for a variety of reasons resulting in the loss of an unknown number of frames, but they are still subject to the aforementioned ordered reliability criteria. This is similar to the guarantee that TCP or SCTP provides for byte streams, and the specification defines a framing system used to parse a byte stream into a sequence of Frames for use in establishing an AMQP Connection over TCP or SCTP.

Containers conduct independent dialogs via named *Sessions*. An AMQP Session is a full-duplex, ordered, ??? providing flow control, and error handling. Sessions send frames over an associated Connection. A single Connection may have multiple independent Sessions, up to a negotiated limit. Both Connections and Sessions are modeled by each peer as *endpoints* that store local and last known remote state regarding the Connection or Session in question.

```
Session A-------+                           +-------Session A
                |                           |
               \|/                         \|/
Session B---> Connection <---------> Connection <---Session B
               /|\                         /|\
                |                           |
Session C-------+                           +-------Session C
```

Sessions provide the context for communication between Link Endpoints. Within a Session, the Link Protocol (defined in the links section) is used to establish Links between local and remote Nodes and to transfer Messages across them. A single Session may simultaneously operate on any number of Links.

```
+-------------+
|    Link     |  Message Transport
+-------------+  (Node to Node)
| name        |
| source      |
| target      |
| timeout     |
+-------------+
     /|\ 0..n
      |
      |
      |
     \|/ 0..1
+------------+
|  Session   |  Frame Transport
+------------+  (Container to Container)
| name       |
+------------+
     /|\ 0..n
      |
      |
      |
     \|/ 1..1
+------------+
| Connection |  Frame Transport
+------------+  (Container to Container)
| principal  |
+------------+
```

A *Frame* is the unit of work carried on the wire. Connections have a negotiated maximum frame size allowing byte streams to be easily defragmented into complete frame bodies representing the independently parsable units formally defined in the frame-bodies section. The following table lists all frame bodies and defines which endpoints handle them.

```
Frame          Connection  Session  Link
========================================
open               H
begin              I          H
attach                        I      H
flow                          I      H
transfer                      I      H
disposition                   I      H
detach                        I      H
end                I          H
close              H
----------------------------------------

Key:
     H: handled by the endpoint

     I: intercepted (endpoint examines
        the frame, but delegates
        further processing to another
        endpoint)
```

# 2 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 a protocol id of zero, followed by three unsigned bytes representing the major, minor, and revision of the protocol version (currently 1, 0, 0). In total this is an 8-octet sequence:

```
  4 OCTETS   1 OCTET   1 OCTET   1 OCTET   1 OCTET
+----------+---------+---------+---------+----------+
|  "AMQP"  |   %d0   |  major  |  minor  | revision |
+----------+---------+---------+---------+----------+
```

Any data appearing beyond the protocol header MUST match the version indicated by the protocol header. If the incoming and outgoing protocol headers do not match, both peers MUST close their outgoing stream and SHOULD read the incoming stream until it is terminated.

The AMQP peer which acted in the role of the TCP client (i.e. the peer that opened the Connection) MUST immediately send its outgoing protocol header on establishment of the TCP Session. The AMQP peer which acted in the role of the TCP server MAY elect to wait until receiving the incoming protocol header before sending its own outgoing protocol header.

Two AMQP peers agree on a protocol version as follows (where the words "client" and "server" refer to the roles being played by the peers at the TCP Connection level):

- When the client opens a new socket Connection to a 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 proceed according to the protocol definition.
- 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%d0.1.0.0       ------------->
                    <-------------       AMQP%d0.1.0.0 (1)
                          ...            *proceed*

AMQP%d0.1.1.0       ------------->
                    <-------------       AMQP%d0.1.0.0 (2)
                                         *TCP CLOSE*

HTTP                ------------->
                    <-------------       AMQP%d0.1.0.0 (3)
                                         *TCP CLOSE*
------------------------------------------------------
  (1) Server accepts Connection for: AMQP, protocol=0,
      major=1, minor=0, revision=0

  (2) Server rejects Connection for: AMQP, protocol=0,
      major=1, minor=1, revision=0, Server responds
      that it supports: AMQP, protocol=0, major=1,
      minor=0, revision=0

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

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

# 3 Framing

Frames are divided into three distinct areas: a fixed width frame header, a variable width extended header, and a variable width frame body.

```
    required        optional        optional
+--------------+-----------------+------------+
| frame header | extended header | frame body |
+--------------+-----------------+------------+
    8 bytes        *variable*      *variable*
```

**frame header:** The frame header is a fixed size (8 byte) structure that precedes each frame. The frame header includes mandatory information required to parse the rest of the frame including size and type information.

**extended header:** The extended header is a variable width area preceeding the frame body. This is an extension point defined for future expansion. The treatment of this area depends on the frame type.

**frame body:** The frame body is a variable width sequence of bytes the format of which depends on the frame type.

## 3.1 Frame Layout

The diagram below shows the details of the general frame layout for all frame types.

```
             +0       +1       +2       +3
           +-----------------------------------+ -.
         0 |                SIZE               |   |
           +-----------------------------------+   |---> Frame Header
         4 |  DOFF  |  TYPE  | <TYPE-SPECIFIC> |   |      (8 bytes)
           +-----------------------------------+ -'
           +-----------------------------------+ -.
         8 |                ...                |   |
           .                                   .   |---> Extended Header
           .          <TYPE-SPECIFIC>          .   |  (DOFF * 4 - 8) bytes
           |                ...                |   |
           +-----------------------------------+ -'
           +-----------------------------------+ -.
    4*DOFF |                                   |   |
           .                                   .   |
           .                                   .   |
           .                                   .   |
           .          <TYPE-SPECIFIC>          .   |---> Frame Body
           .                                   .   |  (SIZE - DOFF * 4) bytes
           .                                   .   |
           .                                   .   |
           .                           ________|   |
           |                ...       |            |
           +--------------------------+          -'
```

**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 of the frame header, extended header, and frame body. The frame is malformed if the size is less than the the size of the required frame header (8 bytes).

**DOFF:** Byte 4 of the frame header is the data offset. This gives the position of the body within the frame. The value of the data offset is unsigned 8-bit integer specifying a count of 4 byte words. Due to the mandatory 8 byte frame header, the frame is malformed if the value is less than 2.

**TYPE:** Byte 5 of the frame header is a type code. The type code indicates the format and purpose of the frame. The subsequent bytes in the frame header may be interpreted differently depending on the type of the frame. A type code of 0x00 indicates that the frame is an AMQP frame.

## 3.2 AMQP Frames

The AMQP frame type defines header bytes 6 and 7 to contain a channel number. The AMQP frame type defines bodies encoded as described types in the AMQP type system.

```
                  type: 0x00 - AMQP frame

               +0       +1       +2       +3
           +-----------------------------------+ -.
         0 |                SIZE               |  |
           +-----------------------------------+  |---> Frame Header
         4 |  DOFF  |  TYPE  |     CHANNEL     |  |      (8 bytes)
           +-----------------------------------+ -'
           +-----------------------------------+ -.
         8 |                ...                |  |
           .                                   .  |---> Extended Header
           .             <IGNORED>             .  |  (DOFF * 4 - 8) bytes
           |                ...                |  |
           +-----------------------------------+ -'
           +-----------------------------------+ -.
    4*DOFF |                                   |  |
           .                                   .  |
           .                                   .  |
           .      Open / Begin / Attach        .  |
           .   Flow / Transfer / Disposition   .  |---> Frame Body
           .      Detach / End / Close         .  |  (SIZE - DOFF * 4) bytes
           .                                   .  |
           .                                   .  |
           .                           ________|  |
           |                ...       |           |
           +--------------------------+          -'
```

A frame with no body may be used to generate artificial traffic as needed to satisfy any negotiated heartbeat interval. Other than resetting the heartbeat timer, an empty AMQP frame has no effect on the recipient.

# 4 Connections

AMQP Connections are divided into a number of unidirectional channels. A Connection Endpoint contains two kinds of channel endpoints: incoming and outgoing. A Connection Endpoint maps incoming frames other than `open` and `close` to an incoming channel endpoint based on the incoming channel number, as well as relaying frames produced by outgoing channel endpoints, marking them with the associated outgoing channel number before sending them.

This requires connection endpoints to contain two mappings. One from incoming channel number to incoming channel endpoint, and one from outgoing channel endpoint, to outgoing channel number.

```
                                  +-------OCHE X: 1
                                  |
                                  +-------OCHE Y: 7
                                  |
<=== Frame[CH=1], Frame[CH=7] <===+

===> Frame[CH=0], Frame[CH=1] ===>+
                                  |
                                  +------>0: ICHE A
                                  |
                                  +------>1: ICHE B

  OCHE: Outgoing Channel Endpoint
  ICHE: Incoming Channel Endpoint
```

Channels are unidirectional, and thus at each connection endpoint the incoming and outgoing channels are completely distinct. Channel numbers are scoped relative to direction, thus there is no causal relation between incoming and outgoing channels that happen to be identified by the "same" number. This means that if a bidirectional endpoint is constructed from an incoming channel endpoint and an outgoing channel endpoint, the channel number used for incoming frames is not necessarily the same as the channel number used for outgoing frames.

```
                                  +-------BIDI/O: 7
                                  |
<=== Frame[CH=1], Frame[CH=7] <===+

===> Frame[CH=0], Frame[CH=1] ===>+
                                  |
                                  +------>1: BIDI/I

  BIDI/I: Incoming half of a single bidirectional endpoint
  BIDI/O: Outgoing half of a single bidirectional endpoint
```

## 4.1 Opening a Connection

Each AMQP Connection begins with an exchange of capabilities and limitations, including the maximum frame size. Prior to any explicit negotiation, the maximum frame size is 4096. After establishing or accepting a TCP Connection and sending the protocol header, each peer must send an `open` frame before sending any other frames. The `open` frame describes the capabilities and limits of that peer. After sending the `open` frame each peer must read its partner's `open` frame 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

                ...
```

## 4.2 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 subsequent frames before receiving the partner's Connection header or `open` frame. This is permitted so long as the pipelined frames 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` frame.

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

                      ...
---------------------------------------------
```

The use of pipelined frames by a peer cannot be distinguished by the peer's partner from non-pipelined use so long as the pipelined frames conform to the partner's capabilities and limitations.

## 4.3 Closing a Connection

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

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

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

## 4.4 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
```

## 4.5 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` frame, but we have not yet received anything.

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

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

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

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

**OPENED:** In this state the the Connection header and the `open` frame have both been sent and received.

**CLOSE_RCVD:** In this state we have received a `close` frame 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 frames 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` frame to our partner. It is illegal to write anything more onto the Connection, however there may still be incoming frames. 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.

## 4.6 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 frames
-  = no frames
** = any frame known a priori to conform to the
     peer's capabilities and limitations
```

# 5 Sessions

Sessions serve three purposes:

- A session maps two channels together to provide bidirectional communication between two endpoints.
- A session provides a grouping for related links allowing transfers along these links to be funneled into a single linear context that matches the expected sequence of processing at the session endpoints.
- Sessions provide transport level flow control for message transfers, thereby permitting the amount of unsettled transfer state to be bounded to the amount required to accomodate the bandwidth delay product of the underlying communication fabric.

TODO: LINK AGGREGATION IS A BETTER TERM TO DESCRIBE WHAT IS GOING ON WITH THE SECOND/THIRD POINTS ABOVE

## 5.1 Establishing a Session

Sessions are established by creating a Session Endpoint, assigning it to an unused channel number, and sending a `begin` announcing the association of the Session Endpoint with the outgoing channel. Upon receiving the `begin` the partner will check the remote-channel field and find it empty. This indicates that the begin is referring to remotely initiated Session. The partner will therefore allocate an unused outgoing channel for the remotely initiated Session and indicate this by sending its own `begin` setting the remote-channel field to the incoming channel of the remotely initiated Session.

The remote-channel field of a `begin` frame MUST be empty for a locally initiated Session, and MUST be set when announcing the endpoint created as a result of a remotely initiated Session.

```
Endpoint                                      Endpoint
=====================================================================
[CH3] BEGIN(name=...,        --------->
            remote-channel=null)
                                    +-- [CH7] BEGIN(name=...,
                                   /                remote-channel=3)
                                  /
                             <---+

                                ...

---------------------------------------------------------------------
```

## 5.2 Ending a Session

Sessions end automatically when the Connection is closed or interrupted. Sessions are explicitly ended when an error occurs on any attached link with an error-mode of "end", or when the initiating endpoint chooses to end the Session. When a Session is explicitly ended an `end` frame is sent announcing the disassociation of the endpoint from its outgoing channel.

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

                                     ...

    [CH3] END(exception=...)     --------->                           (1)
                                        +-- [CH7] END(exception=...)
                                       /
                                      /
(2)                              <---+

                                     ...

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

      (1) At this point the session endpoint is disassociated from
          the outgoing channel on A, and the incoming channel on B.

      (2) At this point the session endpoint is disassociated from
          the outgoing channel on B, and the incoming channel on A.
```

## 5.3 Simultaneous End

Due to the potentially asynchronous nature of Sessions, it is possible that both peers may simultaneously decide to end a Session. If this should happen, it will appear to each peer as though their partner's spontaneously initiated `end` frame is actually an answer to the peers initial `end` frame.

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

                                   ...

    [CH3] END(exception=...)    --+   +-- [CH7] END(exception=...)
(1)                                \ /                                 (2)
                                    x
                                   / \
(3)                             <-+   +->                              (4)

                                   ...

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

      (1) At this point no more frames may be sent by A.

      (2) At this point no more frames may be sent by B.

      (3) At this point Endpoint A is fully ended.

      (4) At this point Endpoint B is fully ended.
```

## 5.4 Session Exceptions

When a Session is unable to process input, it MUST indicate this by issuing an END with an appropriate exception indicating the cause of the error. It MUST then proceed to discard all incoming frames from the remote endpoint until hearing the remote endpoint's corresponding `end` frame.

```
Endpoint                       Endpoint
================================================
FRAME 1            ---------->
FRAME 2            ---------->
FRAME 3            ---+   +--- END(exception=...)
                       \ /
                        x
                       / \
                   <--+   +--> *discarded*
END                ---------->
                       ...
================================================
```

## 5.5 Session States

**UNMAPPED:** In the UNMAPPED state, the Session endpoint is not mapped to any incoming or outgoing channels on the Connection endpoint. In this state an endpoint cannot send or receive frames.

**BEGIN_SENT:** In the BEGIN_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 frames but cannot receive them.

**BEGIN_RCVD:** In the BEGIN_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 frames, but cannot send them.

**MAPPED:** In the MAPPED state, the Session endpoint has both an outgoing channel number and an entry in the incoming channel map. The endpoint may both send and receive frames.

**END_SENT:** In the END_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 frames, but cannot send them.

**END_RCVD:** In the END_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 frames, but cannot receive them.

TODO: UPDATE THIS DIAGRAM TO INCLUDE A DISCARDING STATE AS DEPICTED IN THE SESSION EXCEPTION SECTION

### State Transitions

```
             UNMAPPED<----------------+
                |                     |
        +-------+-------+             |
        |               |             |
S:BEGIN |               | R:BEGIN     |
        |               |             |
       \|/             \|/            |
    BEGIN_SENT      BEGIN_RCVD        |
        |               |             |
        |               |             |
R:BEGIN |               | S:BEGIN     |
        +-------+-------+             |
                |                     |
               \|/                    |
              MAPPED                  |
                |                     |
        +-------+-------+             |
        |               |             |
  S:END |               | R:END       |
        |               |             |
       \|/             \|/            |
     END_SENT        END_RCVD         |
        |               |             |
        |               |             |
  R:END |               | S:END       |
        +-------+-------+             |
                |                     |
                |                     |
                +---------------------+
```

There is no obligation to retain a Session Endpoint when it is in the UNMAPPED state, i.e. the UNMAPPED state is equivalent to a NONEXISTENT state.

# 6 Links

A *Link* provides a unidirectional transport for Messages between two *Link Endpoints*. Link Endpoints interface between the sending/receiving applications, and the Message transport provided by the Link. Link Endpoints therefore come in two flavors, a *Sender* and a *Receiver*. When the sending application submits a Message to the Sender for transport, it also supplies a *delivery-tag* used to track the state of the Message in transit.

The primary responsibility of a Link Endpoint is to maintain a record of the transfer state for each delivery-tag until such a time as it is safe to forget. These are referred to as *unsettled* transfers. When a Link Endpoint forgets the state associated with a delivery-tag, it is considered *settled*. Link Endpoints never spontaneously settle transfers, they only do this when informed by the application that the transfer has been settled. This is usually triggered by the application inspecting the local and/or last known remote transfer state and determining from its value that it is safe to forget.

Link Endpoints may retain additional state when actively communicating (such as the last known remote transfer state), however this state is not necessary for recovery of a Link. Only the unsettled transfer state is necessary for link recovery, and this need not be stored directly. The form of delivery-tags are intentionally left open-ended so that applications can reconstruct the transfer state of a given delivery tag based on application state, thereby reconstructing the necessary state to recover a Link.

TODO: EXPLAIN THE RELATIONSHIP BETWEEN LINKS AND SESSIONS A *Link* is formed when a Sender and a Receiver actively communicate via a Session.

TODO: EXPLAIN SOURCE AND TARGET BETTER The Sender is associated with a *Source*, and the Receiver is associated with a *Target*. The source defines which Messages are supplied to the Sender for transfer, and the target defines how received Messages are disseminated.

## 6.1 Naming a Link

Links are named so that they may be recovered when communication is interrupted. Link names MUST uniquely identify the link amongst all links of the same direction between the two participating containers. Link names are only used when attaching a Link, so they may be arbitrarily long without a significant penalty. TODO: ADD SOME PICTURES TO DEMONSTRATE THE NAMING RULES ARE JUST STANDARD EDGE LABELING CONVENTION FOR GRAPHS

## 6.2 Link Handles

Each Link Endpoint is assigned a numeric handle used by the peer as a shorthand to refer to the Link in all frames that reference the Link (`attach`, `detach`, `flow`, `transfer`, `disposition`). This handle is assigned by the initial `attach` frame and remains in use until the link is detached. The two Endpoints are not required to use the same handle. This means a peer is free to independently chose its handle when a Link Endpoint is associated with the Session.

```
+--------------------+                                  +--------------------+
|      name: Link_1  |                                  |      name: Link_1  |
|     scope: session |                                  |     scope: session |
|    handle: i       |                                  |    handle: j       |
|--------------------|                                  |--------------------|
| direction: IN      |                                  | direction: OUT     |
|    source: NODE_A  |<---+                         +--->|    source: NODE_A  |
|    target: NODE_B  |    |                         |    |    target: NODE_B  |
+--------------------+    |                         |    +--------------------+
                          |                         |
                          |    +---------+          |
          ...         <---+--->| Session |<---+--->          ...
                          |    +---------+          |
                          |                         |
+--------------------+    |                         |    +--------------------+
|      name: Link_N  |    |                         |    |      name: Link_N  |
|     scope: cont    |<---+                         +--->|   durable: cont    |
|    handle: k       |                                  |    handle: l       |
|--------------------|                                  |--------------------|
| direction: OUT     |                                  | direction: IN      |
|    source: NODE_C  |                                  |    source: NODE_C  |
|    target: NODE_D  |                                  |    target: NODE_D  |
+--------------------+                                  +--------------------+
```

## 6.3 Establishing a Link

TODO: SHOULD INDICATE HERE THAT THIS APPLIES EQUALLY WELL TO RESUMING LINKS Links are established by creating a Link Endpoint, assigning it to an unused handle, and sending an `attach` frame carrying the state of the newly created Endpoint. The partner responds with a `attach` frame carrying the state of the corresponding Endpoint, creating and/or mapping the Endpoint to an unused handle if necessary.

```
Peer                                      Partner
================================================================
*create link endpoint*
ATTACH(name=N, handle=1,    ----------> *create link endpoint*
       scope=session,              +--- ATTACH(name=N, handle=2,
       direction=OUT,             /            scope=session,
       source=A,                 /             direction=IN,
       target=B)                /              source=A,
                            <--+               target=B)
                              ...
----------------------------------------------------------------
```

If the partner finds a preexisting Link Endpoint, it MUST respond with the state of the existing endpoint rather than creating a new endpoint. This state may not match the state of the peer's endpoint. In the case where a Link is established between two endpoints with conflicting state, the outgoing endpoint is considered to hold the authoritative version of the source, the incoming endpoint is considered to hold the authoritative version of the target, and the resulting Link state is constructed from the authoritative source and target. Once such a Link is established, either peer is free to continue if the resulting state is acceptable, or if not, `detach`.

```
    Peer                                      Partner
    =================================================================
    *create link endpoint*
    ATTACH(name=N, handle=1,   ----------> *found existing endpoint*
           scope=session,             +--- ATTACH(name=N, handle=2,  (1)
           direction=OUT,            /            scope=session,
           source=A,                /             direction=IN,
           target=B)               /              source=X,
(2)                            <--+               target=Y)
                                   ...
    -----------------------------------------------------------------
      (1) The Link Endpoint already exists, so its state
          may not match the Peer's endpoint.
      (2) At this point the Link is established with source=A,
          target=Y.
```

## 6.4 Resuming a Link

Links may exist beyond their host Session, so it is possible for a Session to terminate and the Link Endpoints to remain. In this case the Link may be resumed the same way a Link is initially established. The existing Link Endpoint is assigned a handle within the new Session, and a `attach` frame is sent with the state of the resuming endpoint.

```
    Peer                                       Partner
    =================================================================
    *existing link endpoint*
    ATTACH(name=N, handle=1,   ----------> *found existing endpoint*
           scope=session,             +--- ATTACH(name=N, handle=2,  (1)
           direction=OUT,            /            scope=session,
           source=X,                /             direction=IN,
           target=Y)               /              source=X,
(2)                            <--+               target=Y)
                                   ...
    -----------------------------------------------------------------
      (1) The Link Endpoint already exists, and its state
          matches the peer's endpoint.
      (2) At this point the Link is reestablished with source=X,
          target=Y.
```

It is possible to resume any Link knowing only the Link name and direction. This is done by creating a new Link Endpoint with an empty source or target for incoming or outgoing Links respectively. The full Link state is then constructed from the authoritative source or target supplied by the other endpoint once the Link is established.

```
    Peer                                   Partner
    ================================================================
    *create link endpoint*
    ATTACH(name=N, handle=1,   ----------> *found existing endpoint*
           scope=session,             +--- ATTACH(name=N, handle=2,  (1)
           direction=OUT,            /            scope=session,
           source=X,                /             direction=IN,
           target=-)               /              source=X,
(2)                            <--+               target=Y)
                                   ...
    ----------------------------------------------------------------
      (1) The Link Endpoint already exists, and its target
          is authoritative.
      (2) At this point the Link is reestablished with source=X,
          target=Y.
```

## 6.5 Closing a Link

TODO: COVER DETACHING A LINK, AND COVER SIMULTANEOUS DETACH/CLOSE CASE A peer closes a Link by sending the `detach` frame with the handle for the specified Link, and the closing flag set. The partner will destroy the corresponding Link endpoint, and reply with its own `detach` frame also setting the closing flag.

```
    Peer                                Partner
    =============================================================
    *create link endpoint*
    ATTACH(name=N, handle=1,    ----------> *create link endpoint*
           scope=session,              +--- ATTACH(name=N, handle=2,
           direction=OUT,             /            scope=session,
           source=A,                 /             direction=IN,
           target=B)                /              source=A,
                              <--+                 target=B)
                                  ...
   *use link*                 <---------> *use link*
                                  ...
    DETACH(handle=1,          ----------> *destroy link endpoint*
           closing=True)
(1) *destroy link endpoint*   <---------- DETACH(handle=2,
                                                 closing=True)
    -------------------------------------------------------------
      (1) At this point both endpoints are destroyed.
```

## 6.6 Flow Control

Once attached, a Link is subject to flow control of Message transfers. Link Endpoints maintain the following flow control state which defines when it is legal to send transfers on an attached Link, as well as indicating when certain interesting conditions occur, such as insufficient transfers to meet the current transfer-limit, or insufficient transfer-limit to send available transfers:

| | |
|---|---|
| **transfer-count:** | The *transfer-count* is initialized to zero when a link is attached, and is incremented whenever message data is sent or received. The *transfer-unit* defines how the transfer-count is incremented. (See the `attach` definition for details.) Only the Sender may independently modify this field. The Receiver's value reflects the last-known value indicated by the Sender. |
| **transfer-limit:** | The *transfer-limit* defines the current maximum legal value of the transfer- |

count. If no transfer-limit is set, then there is currently no maximum legal value for the transfer-count. Only the Receiver can independently modify this field. The Sender's value is always the last-known value indicated by the Receiver.

**attainable-limit:** The *attainable-limit* defines the current maximum value that the transfer-count could reach given a high enough transfer-limit. Only the Sender can independently modify this field. The Receiver's value is always the last-known value indicated by the Sender.

**drain:** The drain flag indicates how the Sender should behave when insufficient transfers are available to meet the current transfer-limit. If set, the Sender will advance the transfer-count to match the transfer-limit and send the flow-state to the Receiver. Only the Receiver can independently modify this field. The Sender's value is always the last-known value indicated by the Receiver. TODO: need to define what to do when drain=True and limit=null, also when transfer-limit is less than transfer-count

If the transfer-count is the same or greater than the transfer-limit, it is illegal to send more transfers until the transfer-limit is increased. Flow control may be disabled entirely if the Receiver sends an empty value for the transfer-limit. If the transfer-limit is reduced by the Receiver when transfers are in-flight, the Receiver MAY either handle the excess transfers normally or detach the Link with a transfer-limit-exceeded error code. TODO: this needs to clearly specify the case where the the transfer-count is less than the current transfer-limit, but the next delivery would cause the transfer-count to exceed the transfer-limit, i.e. we need to decide exactly when fragmentation is expected/mandatory

```
      +----------+                                    +----------+
      |  Sender  |---------------transfer------------>| Receiver |
      +----------+                                    +----------+
       \        / <----------------flow---------------  \        /
        +-------+                                         +-------+
            |
            |
            |
  if transfer-count >= transfer-limit then pause
```

If the Sender's drain flag is set and there are no available transfers, the Sender MUST advance its transfer-count to match the current transfer-limit, and send its updated `flow-state` to the Receiver. TODO: diagram?

The transfer-count, transfer-limit, and attainable-limit, are all absolute values. While the values themselves are conceptually unbounded, they are encoded as 32-bit integers that wraparound and compare according to RFC-1982 serial number arithmetic.

The initial `flow-state` of a Link Endpoint is carried by the `attach` frame. Both the `transfer` frame and the `flow` frame may be used to update it subsequently. When carried on a `transfer` frame, the flow-state includes the current transfer in all its calculations.

```
                                flow-state
                                    |
                                    | carries
+-------------------+               |               +-------------------+
|      Sender       |   .-----------------------.   |     Receiver      |
+-------------------+    attach, transfer, flow     +-------------------+
| transfer-count    |------------------------------>| transfer-count    |
| transfer-limit    |                               | transfer-limit    |
| attainable-limit  |<------------------------------| attainable-limit  |
| drain             |         attach, flow          | drain             |
+-------------------+        '------------'         +-------------------+
                                    |
                                    | carries
                                    |
                                flow-state
```

The flow control semantics defined in this section provide the primitives necessary to implement a wide variety of flow control strategies. Additionally, by manipulating the transfer-limit and drain flag, a Receiver can provide a variety of different higher level behaviors often useful to applications, including synchronous blocking fetch, synchronous fetch with a timeout, asynchronous notifications, and stopping/pausing.

```
            +-----------+                                   +-----------+
            | Receiver  |<-------------transfer-------------|  Sender   |
            +-----------+                                   +-----------+
             \        / -----------------flow--------------> \        /
              +-------+                                       +-------+
                  |
                  |
                  |
    sync-get: flow-state(xfr-limit=xfr-count+1, ...)      ---->
   timed-get: flow-state(xfr-limit=xfr-count+1, ...),
              *wait*,
              flow-state(drain=True, ...)                 ---->
async-notify: flow-state(xfr-limit=xfr-count+delta, ...)  ---->
              - or -
              flow-state(xfr-limit=null, ...)             ---->
        stop: flow-state(xfr-limit=xfr-count, ...)        ---->
```

## 6.7 Synchronous Get

A synchronous get of a transfer from a Link is accomplished by incrementing the transfer-limit, sending the updated `flow-state`, and waiting indefinitely for a `transfer` to arrive.

```
Receiver                                      Sender
=================================================================
                                      ...
flow-state(xfr-limit=xfr-count+1) ---------->
                                        +---- transfer(...)
*block until transfer arrives*         /
                                  <---+
                                      ...
-----------------------------------------------------------------
```

Synchronous get with a timeout is accomplished by incrementing the transfer-limit, sending the updated `flow-state` and waiting for the transfer-count to meet the transfer-limit. When the desired time has elapsed the Receiver then sets the drain flag and sends the newly updated `flow-`

state again, while continuing to wait for the transfer-count to meet the transfer-limit. Even if no transfers are available, this condition will be met promptly because of the drain flag. Once the transfer-count meets the transfer-limit, the Receiver can unambiguously determine whether a transfer has arrived or whether the operation has timed out.

```
    Receiver                                          Sender
    =================================================================
                                          ...
    flow-state(xfr-limit=xfr-count+1) ---------->
  *wait for xfr-count >= xfr-limit*
    flow-state(drain=True)            ---+   +--- transfer(...)
                                          \ /
                                           x
                                          / \
(1)                                   <--+   +-->
(2)                                   <---------- flow-state(...)
                                          ...
    -----------------------------------------------------------------
      (1) If a transfer is available within the timeout, it will
          arrive at this point.
      (2) If a transfer is not available within the timeout, the
          drain flag will ensure that the Sender promptly advances the
          transfer-count to meet transfer-limit.
```

## 6.8 Asynchronous Notification

Asynchronous notification can be accomplished in two ways. If rate limiting of the transfers from a given Link is required, the receiver maintains a target delta between the transfer-limit and the transfer-count for that Link. As transfer arrive on the Link, the actual delta decreases as the transfer-count increases. When the actual delta falls below a threshold, the flow-state may be sent to increase the delta back to the desired target.

```
Receiver                                          Sender
=====================================================================
                                          ...
                                      <---------- transfer(...)
                                      <---------- transfer(...)
flow-state(xfr-limit=xfr-count+delta) ---+   +--- transfer(...)
                                          \ /
                                           x
                                          / \
                                      <--+   +-->
                                      <---------- transfer(...)
                                      <---------- transfer(...)
flow-state(xfr-limit=xfr-count+delta) ---+   +--- transfer(...)
                                          \ /
                                           x
                                          / \
                                      <--+   +-->
                                          ...
---------------------------------------------------------------------
  The incoming transfer rate for the Link is limited by the
  rate at which the Receiver updates the transfer-limit.
```

Alternatively, if there is no need to control the rate of incoming transfers for the Link, the receiver can disable the transfer-limit entirely.

```
Receiver                                  Sender
===============================================================
                                  ...
flow-state(xfr-limit=*empty*) ---------->                           (1)
                                    +---- transfer(...)
                                   /
                              <---+
                              <---------- transfer(...)
                              <---------- transfer(...)
                                  ...
---------------------------------------------------------------
  (1) Once the transfer-limit is disabled, the Sender will
      send transfers at whatever rate they become available.
```

## 6.9 Stopping a Link

Stopping the transfers on a given Link is accomplished by updating the transfer-limit to be less than or equal to the current transfer-count and sending the updated `flow-state`. Some transfers may be in-flight at the time the `flow-state` is sent, so incoming transfers may still arrive on the Link. The echo field of the `flow` frame may be used to request the Sender's `flow-state` be echoed back. This may be used to determine when the Link has finally quiesced.

```
    Receiver                                       Sender
    ================================================================
                                           ...
                                       <---------- transfer(...)
    flow(echo=True,                    ---+   +--- transfer(...)
         flow-state(limit=xfr-count),      \ /
         ...)                               x
                                           / \
(1)                                    <--+   +-->
(2)                                    <---------- flow(...)
                                           ...
    ----------------------------------------------------------------
      (1) In-flight transfers may still arrive until the flow-state
          is updated at the Sender.
      (2) At this point no further transfers will arrive.
```

## 6.10 Transferring a Message

When an application initiates a message transfer, it assigns a delivery-tag used to track the state of the transfer while the message is in transit. A transfer is considered *unsettled* from the point at which it was sent or received until it has been *settled* by the sending/receiving application. Each transfer MUST be identified by a delivery-tag chosen by the sending application. The delivery-tag MUST be unique amongst all transfers that could be considered unsettled by either end of the Link.

The application upon initiating a transfer will supply the sending link endpoint (Sender) with the message data and its associated delivery-tag. The Sender will create an entry in its unsettled map, and send a transfer frame that includes the delivery-tag, its initial state, and its associated message data. For brevity on the wire, the delivery-tag is also associated with a transfer-id assigned by the session. The transfer-id is then used to refer to the delivery-tag in all subsequent interactions on that session. For simplicity the transfer-id is omitted in the following diagrams and the delivery-tag is itself used directly. These diagrams also assume that this interaction takes place in the context of a single established link, and as such omit other details that would be present on the wire in practice such as the channel number, link handle, fragmentation flags, etc, focusing only on the essential aspects of message transfer.

### Initial Transfer

```
                  +-------------------+
                 /       Sender        \
                +-----------------------+
                | unsettled:            |    transfer(delivery-tag=DT, settled=False,
                |   ...                 |             state=S_0, ..., fragments=...)
                |   DT -> (local: S_0,  |------------------------------------------------>
                |          remote: ?)   |
                |   ...                 |
                +-----------------------+
```

Upon receiving the transfer, the receiving link endpoint (Receiver) will create an entry in its own unsettled map and make the transferred message data available to the application to process.

### Initial Receipt

```
                                                                  +-------------------+
                                                                 /      Receiver       \
                                                                +-----------------------+
    transfer(delivery-tag=DT, settled=False,                    | unsettled:            |
             state=S_0, ..., fragments=...)                     |   ...                 |
------------------------------------------------>|              |   DT -> (local: S_1,  |
                                                                |          remote: S_0)|
                                                                |   ...                 |
                                                                +-----------------------+
```

Once notified of the received message data, the application processes the message, indicating the updated transfer state to the link endpoint as desired. Applications may wish to classify transfer states as *terminal* or *non-terminal* depending on whether an endpoint will ever update the state further once it has been reached. In some cases (e.g. large messages or transactions), the receiving application may wish to indicate non-terminal transfer states to the sender. This is done via the disposition frame. TODO: EXTENTS

### Indication of Non-Terminal State

```
                                                  +-------------------+
                                                 /      Receiver       \
                                                +-----------------------+
                                                | unsettled:            |
                                                |   ...                 |
<-----------------------------------------------|   DT -> (local: S_2,  |
    disp(direction=IN, ..., delivery-tag=DT,    |          remote: S_0)|
         settled=False, state=S_2, ...)         |   ...                 |
                                                +-----------------------+
```

Once the receiving application has finished processing the message, it indicates to the link endpoint a *terminal* transfer state that reflects the outcome of the application processing (successful or otherwise). This terminal state is then communicated back to the Sender via the disposition frame.

### Indication of Terminal State

```
                                                +------------------+
                                               /      Receiver      \
                                              +----------------------+
                                              | unsettled:           |
                                              |   ...                |
<---------------------------------------------|   DT -> (local: T_0, |
    disp(direction=IN, ..., delivery-tag=DT,  |          remote: S_0)|
         settled=False, state=T_0, ...)       |   ...                |
                                              +----------------------+
```

Upon receiving the updated transfer state from the Receiver, the Sender will update its view of the remote state and communicate this back to the sending application.

### Receipt of Terminal State

```
  +------------------+
 /       Sender       \
+----------------------+
| unsettled:           |
|   ...                |
|   DT -> (local: S_0, |<---------------------------------------------
|          remote: T_0)|    disp(direction=IN, ..., delivery-tag=DT,
|   ...                |         settled=False, state=T_0, ...)
+----------------------+
```

The sending application will then typically perform some action based on this terminal state and then settle the transfer, causing the Sender to remove the delivery-tag from its unsettled map. The Sender will then send its final transfer state along with an indication that the transfer is settled at the Sender. Note that this amounts to the Sender announcing that it is forever forgetting everything about the delivery-tag in question, and as such it is only possible to make such an announcement once, since after the Sender forgets, it has no way of remembering to make the announcement again. Should this frame get lost due to an interruption in communication, the Receiver will find out that the Sender has settled the transfer upon link recovery at which point the Receiver can derive this fact by examining the unsettled state of the Sender (i.e. what has **not** been forgotten) that is exchanged when the link is reattached.

### Indication of Settlement

```
  +------------------+
 /       Sender       \
+----------------------+
| unsettled:           |    disp(direction=OUT, ..., delivery-tag=DT,
|   ...                |         settled=True, state=T_1, ...)
|   - -> -             |---------------------------------------------->
|   ...                |
+----------------------+
```

When the Receiver finds out that the Sender has settled the transfer, the Receiver will update its view of the remote state to indicate this, and then notify the receiving application.

### Receipt of Settlement

```
                                                   +------------------+
                                                  /      Receiver      \
                                                 +----------------------+
     disp(direction=OUT, ..., delivery-tag=DT,   | unsettled:           |
          settled=True, state=T_1, ...)          |   ...                |
 ----------------------------------------------->|   DT -> (local: S_2, |
                                                 |          remote: - ) |
                                                 |   ...                |
                                                 +----------------------+
```

The application may then perform some final action, e.g. remove the delivery-tag from a set kept for de-duplication, and then notify the Receiver that the transfer is settled. The Receiver will then remove the delivery-tag from its unsettled map. Note that because the Receiver knows that the transfer is already settled at the Sender, it makes no effort to notify the other endpoint that it is settling the transfer.

### Final Settlement

```
                                                   +------------------+
                                                  /      Receiver      \
                                                 +----------------------+
                                                 | unsettled:           |
                                                 |   ...                |
 <-----------------------------------------------|   - -> -             |
                                                 |   ...                |
                                                 +----------------------+
```

This set of exchanges illustrates the basic principals of message transfer. While a transfer is unsettled the endpoints exchange the current state of the transfer. Eventually both endpoints reach a terminal state as indicated by the application. This triggers the other application to take some final action and settle the transfer, and once one endpoint settles, this usually triggers the application at the other endpoint to settle.

This basic pattern can be modified in a variety of ways to achieve different guarantees, for example if the sending application settles the transfer *before* sending it, this results in an *at-most-once* guarantee. The Sender has indicated up front with his initial transmission that he has forgotten everything about this transfer and will therefore make no further attempts to send it. Should this transfer make it to the Receiver, the Receiver clearly has no obligation to respond with updates of the Receiver's transfer state, as they would be meaningless and ignored by the Sender.

### At-Most-Once

```
  +-------------------+
 /       Sender        \
+-----------------------+
| unsettled:            |    transfer(delivery-tag=DT, settled=True,
|   ...                 |             state=T_0, ..., fragments=...)
|   - -> -              |------------------------------------------------>
|   ...                 |
+-----------------------+
```

Similarly, if we modify the basic scenario such that the receiving application chooses to settle immediately upon processing the message rather than waiting for the sender to settle first, we get an *at-least-once* guarantee. If the disposition frame indicated below is lost, then upon link recovery the Sender will not see the delivery-tag in the Receiver's unsettled map and will therefore assume the transfer is lost and resend it, resulting in duplicate processing of the message at the Receiver.

### At-Least-Once

```
                                                   +-------------------+
                                                  /      Receiver       \
                                                 +-----------------------+
                                                 | unsettled:            |
                                                 |   ...                 |
<------------------------------------------------|   - -> -              |
    disp(direction=IN, ..., delivery-tag=DT,     |   ...                 |
         settled=True, state=T_0, ...)           |                       |
                                                 +-----------------------+
```

As you may guess, the scenario presented initially where the sending application settles when the Receiver reaches a terminal state, and the receiving application settles when the Sender settles, results in an *exactly-once* guarantee. More generally if the Receiver settles prior to the Sender, it is possible for duplicate messages to occur, except in the case where the Sender settles before his initial transmission. Similarly, if the Sender settles before the Receiver reaches a terminal state, it is possible for messages to be lost.

TODO: REFERENCE/EXPLAIN RELIABILITY NEGOTIATION ONCE DEFINED The Sender and Receiver policy regarding settling may either be pre-configured for the entire link, thereby allowing for optimized endpoint choices, or may be determined on an ad-hoc basis for each transfer. An application may also choose to settle an endpoint independently of its transfer-state, for example the sending application may choose to settle a message due to the ttl expiring regardless of whether the Receiver has reached a terminal state.

TODO: STATE DIAGRAM FOR TRANSFERS

## 6.11 Resuming Transfers

TODO: DESCRIBE THIS

## 6.12 Message Fragmentation

Messages may be transferred as multiple fragments. Each fragment includes a first flag, last flag, and format-code code that may be used to identify section boundaries and formatting within the Message. For example if Messages are divided into a separate header and body with distinct formats, the first, last, and format-code flags could be used to indicate the boundary between the header and body, and to indicate the distinct formatting of each.

```
  Section 1             Section 2
+-----------+-----------------------------+
|  header   |              body           |
+-----------+-------+-----------+----------+
|    F1     |  F2   |    F3     |    F4    |
+-----------+-------+-----------+----------+

  Fragment | first | last  | format-code
 ===========|=======|=======|==============
     F1    | true  | true  |      0xDB
     F2    | true  | false |      0xCA
     F3    | false | false |      0xCA
     F4    | false | true  |      0xCA
```

Each `transfer` frame may carry an arbitrary number of fragments up to the limit imposed by the maximum frame size. For Messages that are too large to fit within the maximum frame size, additional fragments may be transferred in additional `transfer` frames by setting the more flag on all but the last `transfer` frame.

The sender may indicate an aborted attempt to transfer a Message by setting the abort flag on the last `transfer`. In this case the receiver MUST discard the Message fragments that were transferred prior to the abort.

Messages may be arbitrarily re-fragmented so long as the section boundaries and formatting are still recoverable from the first, last, and format-code fields.

## Outgoing Fragmentation State Diagram

```
              +------------+  S:XFR(M=1,A=0)
       +------|  NOT_SENT  |------+
       |      +------------+      |
       |                          |
       | S:XFR(M=0,A=0)           |
       |                          |     S:XFR(M=1,A=0)
       |                          |      +-----------+
       |                          |      |           |
       |                         \|/    \|/          |
       |                       +------------+        |
       |      +----------------|  SENDING   |--------+
       |      | S:XFR(M=0,A=0) +------------+
       |      |                      |
       |      |                      |
       |      |                      | S:XFR(M=0,A=1)
       |      |                      |
      \|/    \|/                    \|/
    +------------+             +------------+
    |    SENT    |             |  ABORTED   |
    +------------+             +------------+

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

## Incoming Fragmentation State Diagram

```
              +------------+  R:XFR(M=1,A=0)
       +------|  NOT_RCVD  |------+
       |      +------------+      |
       |                          |
       | R:XFR(M=0,A=0)           |
       |                          |     R:XFR(M=1,A=0)
       |                          |      +-----------+
       |                          |      |           |
       |                         \|/    \|/          |
       |                       +------------+        |
       |      +----------------| RECEIVING  |--------+
       |      | R:XFR(M=0,A=0) +------------+
       |      |                      |
       |      |                      |
       |      |                      | R:XFR(M=0,A=1)
       |      |                      |
      \|/    \|/                    \|/
    +------------+             +------------+
    |  RECEIVED  |             |  ABORTED   |
    +------------+             +------------+

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

# 7 Frame-bodies

## 7.1 open (negotiate Connection parameters)

Descriptors: (amqp:open:list, 0x00000001:0x00000201)

Signature: **open**( options: *options*, container-id: *string*, hostname: *string*, max-frame-size: *uint*, channel-max: *ushort*, heartbeat-interval: *ushort*, outgoing-locales: *string*, incoming-locales: *string*, peer-properties: *map*, offered-capabilities: *symbol*, desired-capabilities: *symbol* )

The first frame sent on a connection in either direction MUST contain an Open body. (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: *options* | options map **(optional)** |
| container-id: *string* | the id of the source container **(required)** |
| hostname: *string* | 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: *uint* | 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 error-code. Both peers MUST accept frames of up to 4096 octets large.

| | |
|---|---|
| channel-max: *ushort* | 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 can be simultaneously active on the Connection. 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 error-code.

| | |
|---|---|
| heartbeat-interval: *ushort* | 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.

| | |
|---|---|
| outgoing-locales: *string* | locales available for outgoing text **(multiple)** |

A list of the locales that the peer supports for sending informational text. This includes Connection, Session and Link 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.

| | |
|---|---|
| incoming-locales: *string* | 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 product, "version", giving the version, "pid", giving the process-id of the peer.

| | |
|---|---|
| offered-capabilities: *symbol* | the extension capabilities the sender supports **(multiple)** |

If the receiver of the offered-capabilities requires an extension capability which is not present in the offered-capability list then it MUST close the connection.

| | |
|---|---|
| desired-capabilities: *symbol* | the extension capabilities the sender may use if the receiver supports them **(multiple)** |

The desired-capability list defines which extension capabilities the sender MAY use if the receiver offers them (i.e. they are in the offered-capabilities list received by the sender of the desired-capabilities). If the receiver of the desired-capabilities offers extension capabilities which are not present in the desired-capability list it received, then it can be sure those (undesired) capabilities will not be used on the Connection.

## 7.2 begin (begin a Session on a channel)

| | |
|---|---|
| Descriptors: | (amqp:begin:list, 0x00000001:0x00000202) |
| Signature: | **begin**( options: *options*, remote-channel: *ushort*, name: *string* ) |

Indicate that a Session as begun on the channel.

### Field Details:

| | |
|---|---|
| options: *options* | options map **(optional)** |
| remote-channel: *ushort* | the remote channel for this Session **(optional)** |

If a Session is locally initiated, the remote-channel MUST NOT be set. When an endpoint responds to a remotely initiated Session, the remote-channel MUST be set to the channel on which the remote Session sent the begin.

| | |
|---|---|
| name: *string* | the Session name **(optional)** |

A logical name associated with the session by the sending endpoint. This need not be unique.

## 7.3 attach (attach a Link to a Session)

Descriptors: (amqp:attach:list, 0x00000001:0x00000304)

Signature: **attach**( options: *options*, name: *string*, handle: *handle*, flow-state: *flow-state*, direction: *direction*, source: *, target: *, durable: *boolean*, opening: *boolean*, timeout: *uint*, unsettled: *map*, transfer-unit: *ulong*, error-mode: *error-mode* )

The `attach` frame indicates that a Link Endpoint has been attached to the Session. The opening flag is used to indicate that the Link Endpoint is newly created.

TODO: need to provide capabilities/requirements negotiation around reliability

**Field Details:**

| | |
|---|---|
| options: *options* | options map **(optional)** |
| name: *string* | the name of the link **(required)** |

This name uniquely identifies the link from the container of the source to the container of the target node, e.g. if the container of the source node is A, and the container of the target node is B, the link may be globally identified by the compound name *A*:*B*:*name*. TODO: Add examples/pictures to make this clear. If the remote Link Endpoint does not exist, the peer MUST create a new one or end the session with an error.

| | |
|---|---|
| handle: *handle* | **(required)** |
| flow-state: *flow-state* | **(required)** |
| direction: *direction* | direction of the Link **(optional)** |
| source: * | the source for Messages **(optional)** |

If no source is specified on an outgoing Link, then there is no source currently attached to the Link. A Link with no source will never produce outgoing Messages.

| | |
|---|---|
| target: * | the target for Messages **(optional)** |

If no target is specified on an incoming Link, then there is no target currently attached to the Link. A Link with no target will never permit incoming Messages.

| | |
|---|---|
| durable: *boolean* | indicates that the unassociated Link Endpoint will be durably retained **(optional)** |
| opening: *boolean* | indicates that the Link Endpoint is newly created **(optional)** |
| timeout: *uint* | duration that the unassociated Link Endpoint will be retained **(optional)** |

A Link Endpoint is unassociated when the Connection that carried its most recently attached Session is closed. This timeout is measured in seconds. If the timeout is not set, this indicates that the Link Endpoint will be retained indefinitely.

| | |
|---|---|
| unsettled: *map* | unsettled transfer state **(optional)** |

This is used to indicate any unsettled transfer states when a Link Endpoint is attached to a Session. The map is keyed by delivery-tag with values indicating the transfer state. The local and remote transfer states for a given delivery-tag MUST be compared to resolve any in-doubt transfers. If necessary, transfers MAY be resent, or resumed based on the outcome of this comparison.

| | |
|---|---|
| transfer-unit: *ulong* | the transfer unit **(optional)** |

The transfer-unit defines how the transfer-count is incremented as message data is sent.
The message payload consists of the concatenated fragment payloads. (Note this means concatenating the **contents** of the payload fields, **not** the encoded representation of the payload field.)
The transfer-count is incremented each time a complete message is sent.
The transfer-count is incremented for every transfer-unit bytes of incomplete message payload sent.
This means a message with a payload size of S on a link with a transfer-unit of T will increment the transfer-count (when completely transferred) by S/T rounded up to the nearest integer.
If the transfer-unit is zero or unset, it is taken to be infinitely large, i.e. the transfer-count is only incremented each time a complete message is sent.
The transfer-unit need only be set by the incoming link endpoint. Any value set by an outgoing link endpoint is ignored.

| | |
|---|---|
| error-mode: *error-mode* | the error mode **(optional)** |

## 7.4 flow (update link state)

Descriptors: (amqp:flow:list, 0x00000001:0x00000307)
Signature: **flow**( options: *options*, handle: *handle*, flow-state: *flow-state*, echo: *boolean* )

Updates the flow-state for the specified Link.

**Field Details:**

| | |
|---|---|
| options: *options* | options map **(optional)** |
| handle: *handle* | **(required)** |
| flow-state: *flow-state* | **(required)** |
| echo: *boolean* | request link state from other endpoint **(optional)** |

## 7.5 transfer (transfer a Message)

Descriptors: (amqp:transfer:list, 0x00000001:0x00000309)
Signature: **transfer**( options: *options*, handle: *handle*, flow-state: *flow-state*, delivery-tag: *delivery-tag*, transfer-id: *transfer-number*, settled: *boolean*, state: *, resume: *boolean*, more: *boolean*, aborted: *boolean*, batchable: *boolean*, fragments: *fragment* )

The transfer frame is used to send Messages across a Link. Messages may be carried by a single transfer up to the maximum negotiated frame size for the Connection. Larger Messages may be split across several transfer frames.

**Field Details:**

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

Specifies the Link on which the Message is transferred.

| | |
|---|---|
| flow-state: *flow-state* | **(required)** |
| delivery-tag: *delivery-tag* | **(required)** |

Uniquely identifies the delivery attempt for a given Message on this Link.

| | |
|---|---|
| transfer-id: *transfer-number* | alias for delivery-tag **(required)** |
| settled: *boolean* | **(optional)** |
| state: * | **(optional)** |
| resume: *boolean* | **(optional)** |
| more: *boolean* | indicates that the Message has more content **(optional)** |
| aborted: *boolean* | indicates that the Message is aborted **(optional)** |

Aborted Messages should be discarded by the recipient.

| | |
|---|---|
| batchable: *boolean* | **(optional)** |
| fragments: *fragment* | **(multiple)** |

## 7.6 disposition (inform remote peer of transfer state changes)

Descriptors: (amqp:disposition:list, 0x00000001:0x00009902)
Signature: **disposition**( options: *options*, direction: *direction*, settled: *transfer-number*, unsettled-limit: *transfer-number*, batchable: *boolean*, extents: *extent* )

**Field Details:**

| | |
|---|---|
| options: *options* | **(optional)** |
| direction: *direction* | **(required)** |
| settled: *transfer-number* | low-water mark of unsettled transfers **(required)** |
| unsettled-limit: *transfer-number* | the last transfer which can be sent/received without increasing 'settled' **(required)** |
| batchable: *boolean* | **(optional)** |
| extents: *extent* | **(multiple)** |

## 7.7 detach (detach the Link Endpoint from the Session)

Descriptors: (amqp:detach:list, 0x00000001:0x00000306)
Signature: **detach**( options: *options*, handle: *handle*, closing: *boolean*, exception: *link-error* )

Detach the Link Endpoint from the Session. This un-maps the handle and makes it available for use by other Links.

**Field Details:**

| | |
|---|---|
| options: *options* | options map **(optional)** |
| handle: *handle* | **(required)** |
| closing: *boolean* | **(optional)** |
| exception: *link-error* | error causing the detach **(optional)** |

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

## 7.8 end (end the Session)

Descriptors: (amqp:end:list, 0x00000001:0x00000203)
Signature: **end**( options: *options*, exception: *session-error* )

Indicates that the Session has ended.

**Field Details:**

| | |
|---|---|
| options: *options* | options map **(optional)** |
| exception: *session-error* | error causing the end **(optional)** |

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

## 7.9 close (signal a Connection close)

Descriptors: (amqp:close:list, 0x00000001:0x00000204)
Signature: **close**( options: *options*, exception: *connection-error* )

Sending a close signals that the sender will not be sending any more frames (or bytes of any other kind) on the Connection. Orderly shutdown requires that this frame MUST be written by the sender. It is illegal to send any more frames (or bytes of any other kind) after sending a close frame.

**Field Details:**

| | |
|---|---|
| options: *options* | options map **(optional)** |
| exception: *connection-error* | error causing the close **(optional)** |

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

# 8 Definitions

## 8.1 direction*: ubyte(Link direction)*

**Valid values:**

0 (incoming)

1 (outgoing)

## 8.2 handle*: uint(the handle of a Link)*

An alias established by the `attach` frame and subsequently used by endpoints as a shorthand to refer to the Link in all outgoing frames. 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.

## 8.3 flow-state

| | |
|---|---|
| Descriptors: | (amqp:flow-state:list, 0x00000001:0x00000105) |
| Signature: | **flow-state**( transfer-count: *sequence-no*, transfer-limit: *sequence-no*, attainable-limit: *sequence-no*, drain: *boolean* ) |

If the flow-state is carried in a transfer frame, it includes the containing frame in the calculation.

**Field Details:**

transfer-count: *sequence-no* — the endpoint's transfer-count **(required)**

This field contains the current transfer-count of the endpoint.

transfer-limit: *sequence-no* — the current maximum legal transfer-count **(optional)**

TODO: DOC

attainable-limit: *sequence-no* — the current attainable transfer-limit **(optional)**

TODO: DOC

drain: *boolean* — indicates drain mode **(optional)**

When flow-state is sent from the sender to the receiver, this field contains the actual drain mode of the sender. When flow-state is sent from the receiver to the sender, this field contains the desired drain mode of the receiver.

## 8.4 delivery-tag*: binary*

## 8.5 transfer-number*: sequence-no*

## 8.6 sequence-no*: uint(32-bit RFC-1982 serial number)*

A sequence-no encodes a serial number as defined in RFC-1982. The arithmetic, and operators for these numbers are defined by RFC-1982.

## 8.7 extent

Descriptors: (amqp:extent:list, 0x00000001:0x00099901)
Signature: **extent**( first: *transfer-number*, last: *transfer-number*, handle: *uint*, settled: *boolean*, state: * )

### Field Details:

| | |
|---|---|
| first: *transfer-number* | **(required)** |
| last: *transfer-number* | **(optional)** |
| handle: *uint* | **(optional)** |
| settled: *boolean* | **(optional)** |
| state: * | **(optional)** |

## 8.8 fragment (a Message fragment)

Descriptors: (amqp:fragment:list, 0x00000001:0x00009903)
Signature: **fragment**( first: *boolean*, last: *boolean*, format-code: *uint*, fragment-offset: *ulong*, payload: *binary* )

A Message fragment may contain an entire single section Message, an entire section, or an arbitrary fragment of a single section. A fragment cannot contain data from more than one section.

### Field Details:

first: *boolean* — indicates the fragment is the first in the Section **(optional)**

If this flag is true, then the beginning of the payload corresponds with a section boundary within the Message.

last: *boolean* — indicates the fragment is the last in the Section **(optional)**

If this flag is true, then the end of the payload corresponds with a section boundary within the Message.

format-code: *uint* — indicates the format of the Message section **(optional)**

The format code indicates the format of the current section of the Message. A Message may have multiple sections, and therefore multiple format codes, however the format code is only permitted to change at section boundaries.

| | |
|---|---|
| fragment-offset: *ulong* | the payload offset within the Message **(optional)** |
| payload: *binary* | Message data **(optional)** |

## 8.9 options*: map(options map)*

The options map is used to convey domain or vendor specific optional modifiers on defined AMQP types, e.g. frame bodies. Each key in the map must be of the type `symbol` and all keys except those beginning with the string "x-" are reserved.

### Exceptions:

| | |
|---|---|
| unknown-option | not-implemented *()* |

On receiving a type with an options map containing keys or values which it does not recognise, and for which the key does not begin with the string "x-opt-" an AMQP container MUST detach the link, end the session, or close the connection. TODO: MAKE CLEAR WHEN

## 8.10 error-mode*: ubyte(behaviors for link errors)*

### Valid values:

| | |
|---|---|
| 0 (detach) | When a link error is encountered, the link is detached. |
| 1 (end) | When a link error is encountered, the session is ended, detaching any attached links. |

## 8.11 connection-error (details of a Connection error)

Descriptors: (amqp:connection-error:list, 0x00000001:0x00000101)
Signature: **connection-error**( error-code: *connection-error-code*, description: *string*, error-info: *map* )

### Field Details:

| | |
|---|---|
| error-code: *connection-error-code* | Connection close code **(required)** |

A numeric code indicating the reason for the Connection closure.

| | |
|---|---|
| description: *string* | descriptive text about the exception **(optional)** |

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

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

## 8.12 connection-error-code*: ushort(codes used to indicate the reason for closure)*

**Valid values:**

| | |
|---|---|
| 320 (connection-forced) | An operator intervened to close the Connection for some reason. The client may retry at some later date. |
| 500 (internal-error) | The peer closed the connection because of an internal error. The peer may require intervention by an operator in order to resume normal operations. |
| 501 (framing-error) | A valid frame header cannot be formed from the incoming byte stream. |

## 8.13 session-error (details of a Session error)

Descriptors: (amqp:session-error:list, 0x00000001:0x00000102)

Signature: **session-error**( error-code: *session-error-code*, description: *string*, error-info: *map* )

This struct carries information on an exception which has occurred on the Session.

**Field Details:**

| | |
|---|---|
| error-code: *session-error-code* | error code indicating the type of error **(required)** |
| description: *string* | descriptive text about 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)** |

## 8.14 session-error-code*: ushort(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 made a request 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 frame 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 (frame-invalid) | The frame body could not be decoded. |
| 5006 (resource-limit-exceeded) | The client exceeded its resource allocation. |
| 5030 (not-allowed) | The peer tried to use a frame in a manner that is inconsistent with the semantics defined in the specification. |
| 5031 (illegal-argument) | The frame argument is malformed, i.e. it does not match the specified type. |
| 5040 (not-implemented) | The peer tried to use functionality that is not implemented in its partner. |
| 5041 (internal-error) | An internal error occurred. The peer may require intervention by an operator in order to resume normal operations. |
| 5042 (invalid-argument) | An invalid argument was passed in a frame body, and the operation could not proceed. An invalid argument is not illegal (see illegal-argument), i.e. it matches the type definition; however the particular value is invalid in this context. |

## 8.15 link-error (details of a Link error)

Descriptors: (amqp:link-error:list, 0x00000001:0x00000103)
Signature: **link-error**( error-code: *link-error-code*, description: *string*, error-info: *map* )

**Field Details:**

error-code: *link-error-code* Link error code **(required)**

A numeric code indicating the reason for the Link detach.

description: *string* description text about 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)**

## 8.16 link-error-code*: ushort(codes used to indicate the reason for detach)*

**Valid values:**

| | |
|---|---|
| 320 (detach-forced) | An operator intervened to detach for some reason. |
| 321 (source-destroyed) | The source with which the link was associated has been destroyed. |
| 322 (target-destroyed) | The target with which the link was associated has been destroyed. |
| 401 (not-allowed) | |
| 404 (source-not-found) | The source with which the link was requested to associated does not exist. |
| 405 (target-not-found) | The target with which the link was requested to associated does not exist. |
| 599 (other-error) | The error condition which caused the detach was is cannot be described by any other existing link-error-code. |

### 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*

## 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.

# Book IV - Messaging

# 1 Introduction

The messaging layer builds on top of the concepts described in books II and III. The transport layer defines a number of extension points suitable for use in a variety of different messaging applications. The messaging layer specifies a standardized use of these to provide interoperable messaging capabilities. This standard covers:

- message format
    - properties for the bare message
    - formats for structured and unstructured sections in the bare message
    - headers and footers for the annotated message
- transfer states for messages traveling between nodes
- message states at a node
- sources and targets
    - destructive and non-destructive access to messages from a node
    - filtering of messages from a node
    - default disposition of transfers
    - disposition of orphaned transfers
    - on-demand node creation
- transactional semantics for modifying message states at a node

# 2 Message Format

The term message is used with various connotations in the messaging world. The sender may like to think of the message as an immutable payload handed off to the messaging infrastructure for delivery. The receiver often thinks of the message as not only that immutable payload from the sender, but also various annotations supplied by the messaging infrastructure along the way. To avoid confusion we formally define the term *bare message* to mean the message as supplied by the sender and the term *annotated message* to mean the message as seen at the receiver.

An annotated message consists of the bare message plus areas for annotation at the head and foot of the bare message. There are two classes of annotations: annotations that travel with the message indefinitely, and annotations that are consumed by the next node.

The bare message is divided between standard properties and application data. The standard properties have a defined format and semantics and are visible to the messaging infrastructure. Application data may be AMQP formatted, opaque binary, or a combination of both. AMQP formatted application data is visible to the messaging infrastructure for additional services such as querying by filters.

Altogether the message consists of the following sections:

1. Exactly one `header` section for annotations at the head of the message.
2. Exactly one `properties` section for standard properties in the bare message.
3. Zero or more application data sections.
4. Exactly one `footer` section for annotations at the tail of the message.

```
  Section 0   Section 1    Section 2         Section n-2   Section n-1
+-----------+------------+-----------+-----+-------------+-------------+
|  header   | properties | app-data  | ... |  app-data   |   footer    |
+-----------+------------+-----------+-----+-------------+-------------+
```

Each message section MUST be distinguished and identified with a format-code as per the Message Fragmentation definition in the links section of Book III. The format codes are assigned according to `section-codes`.

## 2.1 section-codes*: uint*

**Valid values:**

| | |
|---|---|
| 0 (header) | Section-code indicating a header section (one of which MUST form the first section of a Message). Sections of this type MUST be of zero length or consist of a single encoded instance of the `header` type. |
| 1 (properties) | Section-code indicating a properties section (one of which MUST form the second section of a Message). Sections of this type MUST be of zero length or consist of a single encoded instance of the `properties` type. |
| 2 (footer) | Section-code indicating a footer section (one of which MUST form the last section of a Message). Sections of this type MUST be of zero length or consist of a single encoded instance of the `footer` type. |
| 3 (map-data) | Section-code indicating a application data section. Sections of this type MUST be of zero length or consist of a single encoded instance of an AMQP the `map`. |
| 4 (list-data) | Section-code indicating a application data section. Sections of this type MUST be of zero length or consist of a single encoded instance of an AMQP the `list`. |
| 5 (data) | Section-code indicating a application data section. Sections of this type consist of opaque binary data (note, in particular, that the section is **not** an instance of the AMQP `binary` type). |

## 2.2 header (transport headers for a Message)

Descriptors: (amqp:header:list, 0x00000001:0x0009801)

Signature: **header**( durable: *boolean*, priority: *ubyte*, transmit-time: *timestamp*, ttl: *ulong*, former-acquirers: *uint*, delivery-failures: *uint*, format-code: *uint*, message-attrs: *message-attributes*, delivery-attrs: *message-attributes* )

The header struct carries information about the transfer of a Message over a specific Link.

**Field Details:**

durable: *boolean* specify durability requirements **(optional)**

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

priority: *ubyte* 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.

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: *ulong* 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.

former-acquirers: *uint* **(optional)**

The number of other DESTRUCTIVE Links to which delivery of this Message was previously attempted unsuccessfully. This does not include the current Link even if delivery to the current Link has been previously attempted.

delivery-failures: *uint* the number of prior unsuccessful delivery attempts **(optional)**

The number of unsuccessful previous attempts to deliver this message. If this value is non-zero it may be taken as an indication that the Message may be a duplicate. The delivery-failures value is initially set to the same value as the Message has when it arrived at the source. It is incremented When: TODO: FIX THIS DEFINITION, I THINK THIS CAN JUST BE DEFINED AS INCREMENTED WHEN INDICATED BY THE OUTCOME NOW

1. the Message has previously been sent from this Node using a Source with a distribution-mode of DESTRUCTIVE which closed without the Message being acknowledged
2. the Message has previously been sent from this Node using a Source with a distribution-mode of DESTRUCTIVE and was subsequently finalized with a disposition of `released` and the delivery-failed field set to true.

format-code: *uint* indicates the format of the Message **(optional)**

message-attrs: *message-attributes* message attributes **(optional)**

The message-attrs map provides an extension point where domain or vendor specific end-to-end attributes can be added to the Message header. As the Message (and therefore the header) passes through a node, the values in the message-attrs map MUST be retained, unless augmented or updated at the node. Further the message-attrs value can be augmented or updated at each node either through node configuration, or when indicated by the `modified` outcome of a message transfer.

delivery-attrs: *message-attributes* — delivery attributes **(optional)**

The delivery-attrs map provides an extension point where domain or vendor specific attributes related only to the current state of the Message at the source, or relating to the current transfer to the target can be added to the Message header. The delivery-attrs value can be augmented or updated at the node either through node configuration, or when indicated by the `modified` outcome of a message transfer.

## 2.3 properties (immutable properties of the Message)

Descriptors: (amqp:properties:list, 0x00000001:0x00009802)

Signature: **properties**( message-id: *binary*, user-id: *binary*, to: *string*, subject: *string*, reply-to: *string*, correlation-id: *binary*, content-length: *ulong*, content-type: *symbol* )

Message properties carry information about the Message.

### Field Details:

message-id: *binary* — 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: *binary* — 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: *string* — 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.

subject: *string* — the subject of the message **(optional)**

A common field for summary information about the Message content and purpose.

reply-to: *string* — the Node to send replies to **(optional)**

The name of the Node to send replies to.

correlation-id: *binary* — 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: *ulong* | length of the combined payload in bytes **(optional)** |

The total size in octets of the combined payload of all fragment that together make the Message.

| | |
|---|---|
| content-type: *symbol* | 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"'.

## 2.4 footer (transport footers for a Message)

Descriptors: (amqp:footer:list, 0x00000001:0x00009803)

Signature: **footer**( message-attrs: *message-attributes*, delivery-attrs: *message-attributes* )

### Field Details:

| | |
|---|---|
| message-attrs: *message-attributes* | message attributes **(optional)** |

The message-attrs map provides an extension point where domain or vendor specific end-to-end attributes can be added to the Message footer. As the Message (and therefore the header) passes through a node, the values in the message-attrs map MUST be retained, unless augmented or updated at the node. Further the message-attrs value can be augmented or updated at each node either through node configuration, or when indicated with the `modified` outcome of a message transfer.

| | |
|---|---|
| delivery-attrs: *message-attributes* | delivery attributes **(optional)** |

The delivery-attrs map provides an extension point where domain or vendor specific attributes related only to the current state of the Message at the source, or relating to the current transfer to the target can be added to the Message footer. The delivery-attrs value can be augmented or updated at the node either through node configuration, or when indicated with the `modified` outcome of a message transfer.

## 2.5 message-attributes*: map(message annotations)*

A map providing an extension point for annotations on message deliveries. All values used as keys in the map MUST be of type symbol. Further if a key begins with the string "x-req-" then the target MUST reject the message unless it understands how to process the supplied key/value.

# 3 Transfer State

The Messaging layer defines a concrete set of transfer states which can be used (via the `disposition` frame) to indicate the state of the message at the receiver. The transfer state includes the number of *bytes-transferred*, the *outcome* of processing at the receiver, and a *txn-id*.

When the outcome is set, this indicates that the state at the receiver is either globally terminal (if no transaction is specified) or provisionally terminal within the specified transaction.

When no outcome is set, a transfer-state is considered non-terminal. In this case the bytes-transferred field may be used to indicate partial progress. Use of this field during link recovery allows the sender to resume the transfer of a large message without retransmitting all the message data.

The following outcomes are formally defined by the messaging layer to indicate the result of processing at the receiver:

- `accepted`: indicates successful processing at the receiver
- `rejected`: indicates an invalid and unprocessable message
- `released`: indicates that the message was not (and will not be) processed
- `modified`: indicates that the message was modified, but not processed

## 3.1 transfer-state (the state of a message transfer)

| | |
|---|---|
| Descriptors: | (amqp:transfer-state:map, 0x00000001:0x00009808) |
| Signature: | **transfer-state**( bytes-transferred: *ulong*, outcome: *, txn-id: * ) |

**Field Details:**

| | |
|---|---|
| bytes-transferred: *ulong* | **(optional)** |
| outcome: * | **(optional)** |
| txn-id: * | **(optional)** |

## 3.2 accepted (the accepted outcome)

| | |
|---|---|
| Descriptors: | (amqp:accepted:map, 0x00000001:0x00009806) |
| Signature: | **accepted**( ) |

## 3.3 rejected (the rejected outcome)

| | |
|---|---|
| Descriptors: | (amqp:rejected:map, 0x00000001:0x00009805) |
| Signature: | **rejected**( reject-properties: *map* ) |

The rejected outcome is used to indicate that an incoming Message is invalid and therefore unprocessable.

**Field Details:**

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

TODO: TURN THIS INTO STANDARD LOOKING EXCEPTION STRUCTURE The map supplied in this field will be placed in any rejected Message headers in the message-attrs map under the key "reject-properties".

## 3.4 released (the released outcome)

Descriptors: (amqp:released:map, 0x00000001:0x00009804)
Signature: **released**( )

The released outcome is used to indicate that a given transfer was not and will not be acted upon.

## 3.5 modified (the modified outcome)

Descriptors: (amqp:modified:map, 0x00000001:0x00009807)
Signature: **modified**( delivery-failed: *boolean*, deliver-elsewhere: *boolean*, message-attrs: *message-attributes*, delivery-attrs: *message-attributes* )

**Field Details:**

delivery-failed: *boolean* — count the transfer as an unsuccessful delivery attempt **(optional)**

If the delivery-failed flag is set, any Messages released MUST have their delivery-failures count incremented.

deliver-elsewhere: *boolean* — prevent redelivery **(optional)**

If the deliver-elsewhere is set, then any Messages released MUST NOT be redelivered to the releasing Link Endpoint.

message-attrs: *message-attributes* — message attributes **(optional)**

Map containing attributes to combine with the existing *message-attrs* held in the Message's header section. Where the existing message-attrs of the Message contain an entry with the same key as an entry in this field, the value in this field associated with that key replaces the one in the existing headers; where the existing message-attrs has no such value, the value in this map is added.

delivery-attrs: *message-attributes* — delivery attributes **(optional)**

Map containing attributes to combine with the existing *delivery-attrs* held in the Message's header section. Where the existing delivery-attrs of the Message contain an entry with the same key as an entry in this field, the value in this field associated with that key replaces the one in the existing headers; where the existing delivery-attrs has no such value, the value in this map is added.

# 4 Message State

The Messaging layer defines a precise set of states for Messages at a Node. The transitions between these states are controlled by the transfer of Messages to/from a Node and the resulting terminal transfer state. Note that the state of a Message at one Node does not affect the state of the same Message at a separate Node.

By default a Message will begin in the AVAILABLE state. Prior to initiating a transfer over a *destructive* Link, the Message will transition to the ACQUIRED state - thus becoming ineligible for transfer to other Links.

A Message will remain ACQUIRED at the sending node until the transfer is settled. The transfer state at the receiver determines how the message transitions when the transfer is settled. If the transfer state at the receiver is not yet known, (e.g. the link endpoint is destroyed before recovery occurs) the *orphan-outcome* of the source is used.

State transitions may also occur spontaneously at the Node. For example if a Message with a ttl expires, the effect of expiry may be (depending on Node type and configuration) to move spontaneously from the AVAILABLE state into the ARCHIVED state. In this case any transfers of the message are settled by the sender regardless of receiver state.

### Message State Transitions

```
                     +------------+
                  +->| AVAILABLE  |
                  |  +------------+
                  |        |
                  |        |
                  |        |
RELEASED/MODIFIED |        | TRANSFER (on a destructive link)
                  |        |
                  |        |
                  |       \|/
                  |  +------------+
                  +--|  ACQUIRED  |
                     +------------+
                           |
                           |
                           |
                           | ACCEPTED/REJECTED
                           |
                           |
                          \|/
                     +------------+
                     |  ARCHIVED  |
                     +------------+
```

# 5 Sources and Targets

The messaging layer defines two concrete types (`source` and `target`) to be used as the *source* and *target* of a link. These types are supplied in the *source* and *target* fields of the `attach` frame when establishing or resuming link. The `source` is comprised of an address (which the container of the outgoing Link Endpoint will resolve to a Node within that container) coupled with properties which determine:

- which messages from the sending Node will be sent on the Link,
- how sending the message affects the state of that message at the sending Node,
- the behaviour of Messages which have been transferred on the Link, but have not yet reached a terminal state at the receiver, when the source is destroyed.

## 5.1 Distribution Modes

The Source defines a distribution-mode. There are two defined distribution-modes: *destructive* and *non-destructive*. The distribution-mode has two related effects on the behaviour of the source.

The *destructive* distribution-mode causes messages transferred from the node to transition to the ACQUIRED state prior to transfer over the link. The *non-destructive* distribution-mode leaves the state of the Message unchanged at the Node.

The distribution-mode of a Source determines how Messages from a Node are distributed among its associated Sources. Messages from a Node MAY be distributed to all associated Sources with a *non-destructive* distribution-mode, but to at most one associated Link with a *destructive* distribution mode. TODO: EXPLAIN THIS BETTER AND POSSIBLY CHANGE NAMES TO ACQUIRING/NON-ACQUIRING

## 5.2 Filtering Messages

A source can restrict the messages transferred from a source by specifying a *filter*. Filters can be thought of as functions which take the message as input and return a boolean value: true if the message will be accepted by the source, false otherwise. A *filter* MUST NOT change its return value for a Message unless the state or annotations on the Message at the Node change (e.g. through an updated transfer state).

Finally the Source MUST NOT resend a Message which has previously been successfully transferred from the Source, i.e. reached an ACCEPTED transfer state at the receiver. For a *destructive* link with a default configuration this is trivially achieved as such an end result will lead to the Message in the ARCHIVED state on the Node, and thus anyway ineligible for transfer. For a *non-destructive* link, state must be retained at the source to ensure compliance. In practice, for nodes which maintain a strict order on Messages at the node, the state may simply be a record of the most recent Message transferred. TODO: THIS IS ODD TO EXPLAIN HERE, THIS ISN'T REALLY ABOUT FILTERING, IT'S ABOUT BROWSING

## 5.3 source

Descriptors: (amqp:source:map, 0x00000001:0x00009701)

Signature: **source**( address: *address*, create: *boolean*, distribution-mode: *distribution-mode*, filter: *filter-set*, orphan-outcome: *, supported-outcomes: *symbol* )

### Field Details:

address: *address* — the address of the source **(optional)**

The address is resolved to a Node in the Container of the sending Link Endpoint. The address of the source MUST be set when sent on a `attach` frame sent by the sending Link Endpoint. When sent by the receiving Link Endpoint the address MUST be set unless the create flag is set, in which case the address MUST NOT be set.

create: *boolean* — 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 unique address. The remote peer will supply the generated address in source field of the corresponding `attach` frame. If the create flag is set, the source address MUST be null. The generated Node will exist until the Source is destroyed. The create flag MUST NOT be set on the source carried by the `attach` sent by the outgoing Link Endpoint.
The algorithm used to produce the address from the Link name, must produce repeatable results. If the Link is durable, generating an address from a given Link name within a given client-id MUST always produce the same result. If the Link is not durable, generating an address from a given Link name within a given Session MUST always produce the same result. The generated address SHOULD include the Link name and Session-name or client-id in some recognizable form for ease of traceability.

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

This field MUST be set by the sending end of the Link. This field MAY be set by the receiving end of the Link to indicate a preference when a Node supports multiple distribution modes.

filter: *filter-set* — a set of predicate to filter the Messages admitted onto the Link **(optional)**

orphan-outcome: * — default outcome for unsettled transfers **(optional)**

Indicates the outcome to be used for transfers that have not reached a terminal state at the receiver when the Link Endpoint is destroyed. The value MUST be a valid outcome (e.g. `released`, or `rejected`). TODO: NEED TO DEFINE THAT THE LINK ENDPOINT HANGS AROUND UNTIL TRANSFERS REFERENCED BY THE SESSION ARE SETTLED

supported-outcomes: *symbol* — outcomes this source supports **(multiple)**

The values in this field are the symbolic descriptors of the outcomes understood by this source. The values MUST be a symbolic descriptor of a valid outcome, e.g. "amqp:accepted:map". TODO: SHOULD THIS ACTUALLY BE A MAP FROM DISTRIBUTION-MODE TO SUPPORTED OUTCOMES?

## 5.4 target

Descriptors: (amqp:target:map, 0x00000001:0x00009702)
Signature: **target**( address: *address*, create: *boolean* )

### Field Details:

| | |
|---|---|
| address: *address* | The address of the target. **(optional)** |

The address is resolved to a Node by the Container of the receiving Link Endpoint. The address of the target MUST be set when sent on a `attach` frame sent by the receiving Link Endpoint. When sent by the sending Link Endpoint the address MUST be set unless the create flag is set, in which case the address MUST NOT be set.

| | |
|---|---|
| create: *boolean* | 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 unique address. The remote peer will supply the generated address in target field of the corresponding `attach` frame. If the create flag is set, the target address MUST be null. The generated Node will exist until the Link is destroyed. The create flag MUST NOT be set on the target carried by the `attach` sent by the incoming Link Endpoint.
The algorithm used to produce the address from the Link name, must produce repeatable results. If the Link is durable, generating an address from a given Link name within a given client-id MUST always produce the same result. If the Link is not durable, generating an address from a given Link name within a given Session MUST always produce the same result. The generated address SHOULD include the Link name and Session-name or client-id in some recognizable form for ease of traceability.

## 5.5 address*: binary(name of the source or target for a Message)*

Specifies the name for a source or target to which Messages are to be transferred to or from. Addresses are expected to be human readable, but are intentionally considered opaque. The format of an address is not defined by this specification.

## 5.6 distribution-mode*: uint(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 |

## 5.7 filter (the predicate to filter the Messages admitted onto the Link)

Descriptors: (amqp:filter:list, 0x00000001:0x00009703)
Signature: **filter**( filter-type: *symbol*, filter: * )

**Field Details:**

| | |
|---|---|
| filter-type: *symbol* | the type of the filter **(required)** |
| filter: * | the filter predicate **(required)** |

## 5.8 filter-set*: map*

A set of named filters. Every key in the map must be of type `symbol`, every value must be of type `filter`. A message will pass through a filter-set if and only if it passes through each of the named filters

# Book V - Transactions

# 1 Transactions

Transactional messaging allows for the coordinated outcome of otherwise independent transfers. This extends to an arbitrary number of transfers spread across any number of distinct links in either direction.

For every transactional interaction, one container acts as the *transactional resource*, and the other container acts as the *transaction controller*. The *transactional resource* performs *transactional work* as requested by the *transaction controller*.

The container acting as the transactional resource defines a special target that functions as a `coordinator`. The *transaction controller* establishes a link to this target and allocates a container scoped transaction id (txn-id) by sending a message whose body consists of the `declare` type. This txn-id is then used by the controller to associate work with the transaction. The controller will subsequently end the transaction by sending a `discharge` message to the controller. Should the controller fail to perform the `declare` or `discharge` as specified in the message, the message will be rejected with a `txn-error-code` indicating the failure condition that occurred. TODO: DEFINE HOW ERROR-CODE IS CARRIED BY REJECT OUTCOME (THIS WILL BE TAKEN CARE OF BY RATIONALIZING ERROR CODES), AND ADD DIAGRAM

Transactional work is described in terms of the message states defined in the message-state section of the messaging book. Transactional work is formally defined to be composed of the following operations:

- *posting* a message at a node, i.e. making it *available*
- *acquiring* a message at a node, i.e. transitioning it to *acquired*
- *retiring* a message at a node, i.e. transitioning it to *archived*

The transactional resource performs these operations when triggered by the transaction controller:

- *posting* messages is initiated by incoming transfer frames
- *acquiring* messages is initiated by incoming flow frames
- *retiring* messages is initiated by incoming disposition frames

In each case, it is the responsibility of the transaction controller to identify the transaction with which the requested work is to be associated. This is done by indicating the txn-id in the transfer-state associated with a given message transfer. The transfer-state is carried by both the transfer and the disposition frames allowing both the *posting* and *retiring* of messages to be associated with a transaction. TODO: ADD DIAGRAM

In the case of the flow frame, the transactional work is not necessarily directly initiated or entirely determined when the flow frame arrives at the resource, but may in fact occur at some later point and in ways not necessarily anticipated by the controller. To accomodate this, the resource associates an additional piece of state with outgoing link endpoints, an optional *txn-id* that identifies the transaction with which *acquired* messages will be associated. This state is determined by the controller by specifying a *txn-id* entry in the *options* map of the flow frame. When a transaction is discharged, the *txn-id* of any link endpoints will be cleared. TODO: ADD DIAGRAM

The transfer, disposition, and flow frames may travel in either direction, i.e. both from the controller to the resource and from the resource to the controller. When these frames travel from the controller to the resource, any embedded txn-ids are requesting that the resource assigns transactional work to the indicated transaction. When traveling in the other direction, from resource to controller, the transfer and disposition frames indicate work performed, and the txn-ids included MUST correctly indicate with which (if any) transaction this work is associated. In the case of the flow frame traveling from resource to controller, the txn-id does not indicate work that has been performed, but indicates with which transaction future transfers from that link will be performed. TODO: ADD DIAGRAM

# 2 Coordination

## 2.1 coordinator (target for communicating with a transaction coordinator)

Descriptors: (amqp:coordinator:map, 0x00000001:0x00000300)
Signature: **coordinator**( capabilities: *symbol*, multi-txns-per-ssn: *boolean*, multi-ssns-per-txn: *boolean*, multi-conns-per-txn: *boolean* )

The coordinator type defines a special target used for establishing a link with a transaction coordinator. The transaction controller indicates the desired capabilities using the fields specified in this type. The transaction coordinator will indicate the actual capabilities using these same fields.

**Field Details:**

| | |
|---|---|
| capabilities: *symbol* | **(multiple)** |
| multi-txns-per-ssn: *boolean* | **(optional)** |
| multi-ssns-per-txn: *boolean* | **(optional)** |
| multi-conns-per-txn: *boolean* | **(optional)** |

## 2.2 declare (message body for declaring a transaction id)

Descriptors: (amqp:declare:list, 0x00000001:0x00000301)
Signature: **declare**( options: *options*, global-id: *binary* )

The declare type defines the message body sent to the coordinator to declare a transaction. The txn-id allocated for this transaction is chosen by the transaction controller and identified in the transfer-state associated with the containing message.

**Field Details:**

| | |
|---|---|
| options: *options* | options map **(optional)** |
| global-id: *binary* | global transaction id **(optional)** |

Specifies that the txn-id allocated by this declare will associated work with the indicated global transaction. If not set, the allocated txn-id will associated work with a local transaction.

## 2.3 discharge (message body for discharging a transaction)

Descriptors: (amqp:discharge:list, 0x00000001:0x00000303)
Signature: **discharge**( options: *options*, fail: *boolean* )

The discharge type defines the message body sent to the coordinator to indicate that the txn-id is no longer in use. If the transaction is not associated with a global-id, then this also indicates the disposition of the local transaction.

**Field Details:**

| | |
|---|---|
| options: *options* | options map **(optional)** |
| fail: *boolean* | indicates the transaction should be rolled back **(optional)** |

If set, this flag indicates that the work associated with this transaction has failed, and the controller wishes the transaction to be rolled back. If the transaction is associated with a global-id this will render the global transaction rollback-only. If the transaction is a local transaction, then this flag controls whether the transaction is committed or aborted when it is discharged.

## 2.4 txn-error-code*: ushort(codes used to indicate the reason for closure)*

**Valid values:**

| | |
|---|---|
| 1 (internal-error) | An internal error occurred. |
| 2 (unknown-txn-id) | The specified txn-id does not exist. |
| 3 (transaction-rollback) | The transaction was rolled back for an unspecified reason. |
| 4 (transaction-timeout) | The work represented by this transaction took too long. |

# Book VI - Security

# 1 Security Layers

Security Layers are used to establish an authenticated and/or encrypted transport over which regular AMQP traffic can be tunneled. Security Layers may be tunneled over one another (for instance a Security Layer used by the peers to do authentication may be tunneled over a Security Layer established for encryption purposes).

The framing and protocol definitions for security layers are expected to be defined externally to the AMQP specification as in the case of TLS. An exception to this is the SASL security layer which depends on its host protocol to provide framing. Because of this we define the frames necessary for SASL to function in the sasl section below. When a security layer terminates (either before or after a secure tunnel is established), the TCP Connection MUST be closed by first shutting down the outgoing stream and then reading the incoming stream until it is terminated.

# 2 TLS

To establish a TLS tunnel, each peer MUST start by sending a protocol header. The protocol header consists of the upper case ASCII letters "AMQP" followed by a protocol id of one, followed by three unsigned bytes representing the major, minor, and revision of the specification version (currently MAJOR, MINOR, REVISION). In total this is an 8-octet sequence:

```
  4 OCTETS   1 OCTET   1 OCTET   1 OCTET   1 OCTET
+----------+---------+---------+---------+----------+
|  "AMQP"  |   %d1   |  major  |  minor  | revision |
+----------+---------+---------+---------+----------+
```

Other than using a protocol id of one, the exchange of TLS tunnel headers follows the same rules specified in the version negotiation section of the transport specification (See version-negotiation).

The following diagram illustrates the interaction involved in creating a TLS Security Layer:

```
TCP Client                 TCP Server
=========================================
AMQP%d1.1.0.0  --------->
               <---------  AMQP%d1.1.0.0
                    :
                    :
            <TLS negotiation>
                    :
                    :
AMQP%d0.1.0.0  --------->                 (over TLS secured connection)
               <---------  AMQP%d0.1.0.0
         open  --------->
               <---------  open
```

When the use of the TLS Security Layer is negotiated, the following rules apply:

- The TLS client peer and TLS server peer are determined by the TCP client peer and TCP server peer respectively.
- The TLS client peer SHOULD use the server name indication extension as described in RFC-4366.
- The TLS client MUST validate the certificate presented by the TLS server.

# 3 SASL

To establish a SASL tunnel, each peer MUST start by sending a protocol header. The protocol header consists of the upper case ASCII letters "AMQP" followed by a protocol id of two, followed by three unsigned bytes representing the major, minor, and revision of the specification version (currently MAJOR, MINOR, REVISION). In total this is an 8-octet sequence:

```
  4 OCTETS   1 OCTET   1 OCTET   1 OCTET   1 OCTET
+----------+---------+---------+---------+----------+
|  "AMQP"  |   %d2   |  major  |  minor  | revision |
+----------+---------+---------+---------+----------+
```

Other than using a protocol id of two, the exchange of SASL tunnel headers follows the same rules specified in the version negotiation section of the transport specification (See version-negotiation).

The following diagram illustrates the interaction involved in creating a SASL Security Layer:

```
TCP Client                 TCP Server
=========================================
AMQP%d2.1.0.0  --------->
               <---------  AMQP%d2.1.0.0
                    :
                    :
            <SASL negotiation>
                    :
                    :
AMQP%d0.1.0.0  --------->                (over SASL secured connection)
               <---------  AMQP%d0.1.0.0
         open  --------->
               <---------  open
```

## 3.1 SASL Negotiation

The peer acting as the SASL Server must announce supported authentication mechanisms using the `sasl-mechanisms` frame. The partner must then choose one of the supported mechanisms and initiate a sasl exchange.

### SASL Exchange

```
SASL Client       SASL Server
================================
              <-- SASL-MECHANISMS
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 MUST correspond to the TCP client and server respectively.

## 3.2 Security Frame Bodies

### 3.3 sasl-mechanisms (advertise available sasl mechanisms)

Descriptors: (amqp:sasl-mechanisms:list, 0x00000001:0x00000701)
Signature: **sasl-mechanisms**( options: *map*, sasl-server-mechanisms: *string* )

Advertises the available SASL mechanisms that may be used for authentication.

#### Field Details:

| | |
|---|---|
| options: *map* | options map **(optional)** |
| sasl-server-mechanisms: *string* | 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 list may be empty or absent. The server mechanisms are ordered in decreasing level of preference.

### 3.4 sasl-init (initiate sasl exchange)

Descriptors: (amqp:sasl-init:list, 0x00000001:0x00000702)
Signature: **sasl-init**( options: *map*, mechanism: *string*, initial-response: *binary* )

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

#### Field Details:

| | |
|---|---|
| options: *map* | options map **(optional)** |
| mechanism: *string* | 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: *binary* | 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.

## 3.5 sasl-challenge (security mechanism challenge)

Descriptors: (amqp:sasl-challenge:list, 0x00000001:0x00000703)
Signature: **sasl-challenge**( options: *map*, challenge: *binary* )

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

### Field Details:

| | |
|---|---|
| options: *map* | options map **(optional)** |
| challenge: *binary* | security challenge data **(required)** |

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

## 3.6 sasl-response (security mechanism response)

Descriptors: (amqp:sasl-response:list, 0x00000001:0x00000704)
Signature: **sasl-response**( options: *map*, response: *binary* )

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

### Field Details:

| | |
|---|---|
| options: *map* | options map **(optional)** |
| response: *binary* | 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.

## 3.7 sasl-outcome (indicates the outcome of the sasl dialog)

Descriptors: (amqp:sasl-outcome:list, 0x00000001:0x00000705)
Signature: **sasl-outcome**( options: *map*, code: *sasl-code*, additional-data: *binary* )

This frame indicates the outcome of the SASL dialog. Upon successful completion of the SASL dialog the Security Layer has been established, and the peers must exchange protocol headers to either start a nested Security Layer, or to establish the AMQP Connection.

**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: *binary* | 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.

## 3.8 sasl-code*: ubyte(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. |