AMQP v1.0 (revision 0)

05 Apr 2011

Confidential Draft

URL:	`https://www.amqp.org/svn/amqp/trunk`
Revision:	`1239M`

Copyright Notice

Trademarks

Link to full AMQP specification

http://www.amqp.org/confluence/display/AMQP/AMQP+Specification

Contents

Introduction

Overview

The Advanced Message Queuing Protocol 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.

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*.

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 • Consisting of business messages that are tamper proof • Supporting message durability independent of receivers being connected, and • 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

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	Overview of the AMQP Layering
Book II	Defines the AMQP Type System
Book III	Defines the AMQP Transport Layer
Book IV	Defines the AMQP Messaging Layer
Book V	Defines the AMQP Transaction Layer
Book VI	Defines the AMQP Security Layers

Book 1

Layering

```
+-----+-----+-------------------------------------+
|     |     |    Broker     |     Application     |
|     |     |---------------|---------------------|
|  E  |  T  | Queue | Topic | Producer | Consumer |
|  n  |  y  +=====================================+
|  c  |  p  |    Messaging    |    Transactions   |
|  o  |  e  +=====================================+
|  d  |  s  |               Link                  |
|  i  |     |-------------------------------------|
|  n  |     |              Session                |
|  g  |     +=====================================+
|     |     |              Channel                |
|     |     |-------------------------------------|
|     |     |             Connection              |
+-----+-----+=====================================+
            +=====================================+
            |              Framing                |
            |-------------------------------------|
            |         Version Negotiation         |
            +=====================================+
            +=====================================+
            |              Security               |
            |-------------------------------------|
            |        Security Negotiation         |
            +=====================================+
            +=====================================+
            |  TCP  |  SCTP  |  RDMA  |  UDP      |
            +-------+--------+--------+-----------+
            |               ...                   |
            +-------------------------------------+
```

Book 2

Types

2.1 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.

2.1.1 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)
decimal32	32-bit decimal number (IEEE 754-2008 decimal32)
decimal64	64-bit decimal number (IEEE 754-2008 decimal64)
decimal128	128-bit decimal number (IEEE 754-2008 decimal128)
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	symbolic values from a constrained domain

list	a sequence of polymorphic values
map	a polymorphic mapping from distinct keys to values

2.1.2 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 list of mappings between AMQP types and programming language types can be found here:

2.1.3 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.

2.1.4 Descriptor Values

Descriptor values other than symbolic (`symbol`) or numeric (`ulong`) are, while not syntactically invalid, reserved - this includes numeric types other than `ulong`. To allow for users of the type system to define their own descriptors without collision of descriptor values, an assignment policy for symbolic and numeric descriptors is given below.

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

$(domain\text{-}id << 32) \mid descriptor\text{-}id$

2.2 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.

```
       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
```

Figure 2.1: Primitive Format Code (String)

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.

```
            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

     (Note: this example shows a string-typed descriptor, which should be
      considered reserved)
```

Figure 2.2: Described Format Code (URL)

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 = 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
```

Figure 2.3: Constructor BNF

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.

2.2.1 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

Type: `null`

```
<type name="null" class="primitive"/>
```

Encoding	Code	Category	Description
	0x40	fixed-width, 0 byte value	*the null value*

Type: `boolean`

```
<type name="boolean" class="primitive"/>
```

Encoding	Code	Category	Description
`true`	0x41	fixed-width, 0 byte value	*the boolean value true*
`false`	0x42	fixed-width, 0 byte value	*the boolean value false*

Type: `ubyte`

```
<type name="ubyte" class="primitive"/>
```

Encoding	Code	Category	Description
	0x50	fixed-width, 1 byte value	*8-bit unsigned integer*

Type: `ushort`

```
<type name="ushort" class="primitive"/>
```

Encoding	Code	Category	Description
	0x60	fixed-width, 2 byte value	*16-bit unsigned integer in network byte order*

Type: `uint`

```
<type name="uint" class="primitive"/>
```

Encoding	Code	Category	Description
	0x70	fixed-width, 4 byte value	*32-bit unsigned integer in network byte order*
`smalluint`	0x52	fixed-width, 1 byte value	*unsigned integer value in the range 0-255*

Type: `ulong`

`<type name="ulong" class="primitive"/>`

Encoding	Code	Category	Description
	0x80	fixed-width, 8 byte value	*64-bit unsigned integer in network byte order*
`smallulong`	0x53	fixed-width, 1 byte value	*unsigned long value in the range 0-255*

Type: `byte`

`<type name="byte" class="primitive"/>`

Encoding	Code	Category	Description
	0x51	fixed-width, 1 byte value	*8-bit two's-complement integer*

Type: `short`

`<type name="short" class="primitive"/>`

Encoding	Code	Category	Description
	0x61	fixed-width, 2 byte value	*16-bit two's-complement integer in network byte order*

Type: `int`

`<type name="int" class="primitive"/>`

Encoding	Code	Category	Description
	0x71	fixed-width, 4 byte value	*32-bit two's-complement integer in network byte order*
`smallint`	0x54	fixed-width, 1 byte value	*unsigned integer value in the range -128-127*

Type: `long`

`<type name="long" class="primitive"/>`

Encoding	Code	Category	Description
	0x81	fixed-width, 8 byte value	*64-bit two's-complement integer in network byte order*
`smalllong`	0x55	fixed-width, 1 byte value	*unsigned long value in the range -128-127*

Type: `float`

```
<type name="float" class="primitive"/>
```

Encoding	Code	Category	Description
`ieee-754`	0x72	fixed-width, 4 byte value	*IEEE 754-2008 binary32*

Type: `double`

```
<type name="double" class="primitive"/>
```

Encoding	Code	Category	Description
`ieee-754`	0x82	fixed-width, 8 byte value	*IEEE 754-2008 binary64*

Type: `decimal32`

```
<type name="decimal32" class="primitive"/>
```

Encoding	Code	Category	Description
`ieee-754`	0x74	fixed-width, 4 byte value	*IEEE 754-2008 decimal32 using the Binary Integer Decimal encoding*

Type: `decimal64`

```
<type name="decimal64" class="primitive"/>
```

Encoding	Code	Category	Description
`ieee-754`	0x84	fixed-width, 8 byte value	*IEEE 754-2008 decimal64 using the Binary Integer Decimal encoding*

Type: `decimal128`

```
<type name="decimal128" class="primitive"/>
```

Encoding	Code	Category	Description
`ieee-754`	0x94	fixed-width, 16 byte value	*IEEE 754-2008 decimal128 using the Binary Integer Decimal encoding*

Type: `char`

```
<type name="char" class="primitive"/>
```

Encoding	Code	Category	Description
`utf32`	0x73	fixed-width, 4 byte value	*a UTF-32BE encoded unicode character*

Type: `timestamp`

`<type name="timestamp" class="primitive"/>`

Encoding	Code	Category	Description
`ms64`	0x83	fixed-width, 8 byte value	*64-bit signed integer representing milliseconds since the unix epoch*

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.

Type: `uuid`

`<type name="uuid" class="primitive"/>`

Encoding	Code	Category	Description
	0x98	fixed-width, 16 byte value	*UUID as defined in section 4.1.2 of RFC-4122*

2.2.2 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
```

Type: `binary`

`<type name="binary" class="primitive"/>`

Encoding	Code	Category	Description
`vbin8`	0xa0	variable-width, 1 byte size	*up to* 2^8 *- 1 octets of binary data*
`vbin32`	0xb0	variable-width, 4 byte size	*up to* 2^{32} *- 1 octets of binary data*

Type: `string`

```
<type name="string" class="primitive"/>
```

A string represents a sequence of unicode characters as defined by the Unicode V6.0.0 standard (see http://www.unicode.org/versions/Unicode6.0.0).

Encoding	**Code**	**Category**	**Description**
`str8-utf8`	0xa1	variable-width, 1 byte size	*up to* 2^8 *- 1 octets worth of UTF-8 unicode (with no byte order mark)*
`str8-utf16`	0xa2	variable-width, 1 byte size	*up to* 2^8 *- 1 octets worth of UTF-16BE unicode (with no byte order mark)*
`str32-utf8`	0xb1	variable-width, 4 byte size	*up to* 2^{32} *- 1 octets worth of UTF-8 unicode (with no byte order mark)*
`str32-utf16`	0xb2	variable-width, 4 byte size	*up to* 2^{32} *- 1 octets worth of UTF-16BE unicode (with no byte order mark)*

Type: `symbol`

```
<type name="symbol" class="primitive"/>
```

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.

Encoding	**Code**	**Category**	**Description**
`sym8`	0xa3	variable-width, 1 byte size	*up to* 2^8 *- 1 seven bit ASCII characters representing a symbolic value*
`sym32`	0xb3	variable-width, 4 byte size	*up to* 2^{32} *- 1 seven bit ASCII characters representing a symbolic value*

2.2.3 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
```

Type: `list`

`<type name="list" class="primitive"/>`

Encoding	Code	Category	Description
`list8`	0xc0	variable-width, 1 byte size	*up to 2^8 - 1 list elements with total size less than 2^8 octets*
`list32`	0xd0	variable-width, 4 byte size	*up to 2^{32} - 1 list elements with total size less than 2^{32} octets*
`array8`	0xe0	variable-width, 1 byte size	*up to 2^8 - 1 array elements with total size less than 2^8 octets*
`array32`	0xf0	variable-width, 4 byte size	*up to 2^{32} - 1 array elements with total size less than 2^{32} octets*

Type: `map`

`<type name="map" class="primitive"/>`

Encoding	Code	Category	Description
`map8`	0xc1	variable-width, 1 byte size	*up to 2^8 - 1 octets of encoded map data*
	Up to 2^8 - 1 octets of encoded map data. A map is encoded as a compound value where the constituent elements form alternating key value pairs. (See Figure 2.2.3).		
`map32`	0xd1	variable-width, 4 byte size	*up to 2^{32} - 1 octets of encoded map data*
	Up to 2^{32} - 1 octets of encoded map data. See map8 above for a definition of encoded map data.		

```
 item 0   item 1      item n-1    item n
+-------+-------+----+---------+---------+
| key 1 | val 1 | .. | key n/2 | val n/2 |
+-------+-------+----+---------+---------+
```

Figure 2.4:

2.2.4 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
```

2.2.5 List Of Encodings

Type	Encoding	Code	Category	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
uint	smalluint	0x52	fixed/1	unsigned integer value in the range 0-255
ulong		0x80	fixed/8	64-bit unsigned integer in network byte order
ulong	smallulong	0x53	fixed/1	unsigned long value in the range 0-255
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
int	smallint	0x54	fixed/1	unsigned integer value in the range -128-127
long		0x81	fixed/8	64-bit two's-complement integer in network byte order
long	smalllong	0x55	fixed/1	unsigned long value in the range -128-127
float	ieee-754	0x72	fixed/4	IEEE 754-2008 binary32
double	ieee-754	0x82	fixed/8	IEEE 754-2008 binary64
decimal32	ieee-754	0x74	fixed/4	IEEE 754-2008 decimal32 using the Binary Integer Decimal encoding
decimal64	ieee-754	0x84	fixed/8	IEEE 754-2008 decimal64 using the Binary Integer Decimal encoding
decimal128	ieee-754	0x94	fixed/16	IEEE 754-2008 decimal128 using the Binary Integer Decimal encoding
char	utf32	0x73	fixed/4	a UTF-32BE encoded unicode character
timestamp	ms64	0x83	fixed/8	64-bit signed integer representing milliseconds since the unix epoch
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 (with no byte order mark)
string	str8-utf16	0xa2	variable/1	up to 2^8 - 1 octets worth of UTF-16BE unicode (with no byte order mark)
string	str32-utf8	0xb1	variable/4	up to 2^{32} - 1 octets worth of UTF-8 unicode (with no byte order mark)
string	str32-utf16	0xb2	variable/4	up to 2^{32} - 1 octets worth of UTF-16BE unicode (with no byte order mark)
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
map	map32	0xd1	compound/4	up to 2^{32} - 1 octets of encoded map data

2.3 Composite Types

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

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

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

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

  <field name="title" type="string" mandatory="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>
```

Figure 2.5: Example Composite Type

The *mandatory* 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 multiple element values are permitted in the representation. A single element of the type specified in the field description is always permitted, as is a null element value. Multiple values are represented by the use of a described list (of any valid encoding of the list type). The descriptor for the list MUST be the boolean value *true*, and every element in the list MUST be of the type matching the field description. Note that a null value and a described list (with descriptor true) of zero elements both describe an absence of a value and should be treated as semantically identical. Lists described with the value true (i.e. the Multiple type) MUST NOT contain null entries. A field which is defined as both multiple and mandatory MUST contain at least one value (i.e. for such a field both *null* and a Multiple list with zero entries are invalid).

AMQP composite types may be encoded either as a described list or a described map. In general a specific composite 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.)

2.3.1 List Encoding

When encoding AMQP composite values as a described list, each element in the list is positionally correlated with the fields listed in the composite 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 descriptor of the list indicates the specific composite type being represented.

The described list shown below is an example composite 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.

```
             constructor                  list representation of a book
                  |                                   |
+-----------------+-------------------+ +-------------+---------------+
|                                     | |                             |
0x00 0xA3 0x11 "example:book:list" 0xC0 0x42 0x03  title  authors  isbn
      |                 |                            |       |       |
      |      identifies composite type               |       |       |
      |                                              |       |     0x40
    sym8                      +----------------------+       |       |
  (symbol)                    |                              |  null value
               +--------------+----------------+             |
               |                               |             |
               0xA1 0x15 "AMQP for & by Dummies"             |
                                                             |
+------------------------------------------------------------+-----+
|                                                                  |
0x00 0x41 0xE0 0x25 0x02 0xA1 0x0E "Rob J. Godfrey" 0x13 "Rafael H. Schloming"
      |         |    |    |   |                   | |                        |
      |       size   |    |   +---------+---------+ +-----------+------------+
      |              |    |             |                       |
    true           count  |        first element          second element
(Multiple type)           |
                  element constructor
```

Figure 2.6: Example Composite Value

2.3.2 Map Encoding

When encoding AMQP composite values as a described map, the set of valid keys is restricted to symbols identifying the field names in the composite 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 descriptor of the map indicates the specific composite type being represented.

Book 3

Transport

3.1 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. Links attach to a Node at a *Terminus*. There are two kinds of Terminus: *Sources* and *Targets*. A Terminus is responsible for tracking the state of a particular stream of incoming or outgoing messages. Sources track outgoing messages and Targets track incoming messages. Messages may only travel along a Link if they meet the entry criteria at the Source.

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 the Source and Target.

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

            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$^{\text{th}}$ 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.

An AMQP Connection is divided into a negotiated number of independent unidirectional *Channels*. Each Frame is marked with the Channel number indicating its parent Channel, and the Frame sequence for each Channel is multiplexed into a single Frame sequence for the Connection.

An AMQP *Session* correlates two unidirectional Channels to form a bidirectional, sequential conversation between two Containers. A single Connection may have multiple independent Sessions active simultaneously, up to the negotiated Channel 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<------+                           +------>Session
(ICH=1, OCH=1)    |                           |    (ICH=1, OCH=1)
                 \|/                         \|/
    Session<--> Connection <---------> Connection <-->Session
(ICH=2, OCH=3)   /|\                         /|\   (ICH=3, OCH=2)
                  |                           |
    Session<------+                           +------>Session
(ICH=3, OCH=2)                                     (ICH=2, OCH=3)

        Key: ICH -> Input Channel, OCH -> Output Channel
```

Figure 3.1: Session & Connection Endpoints

Sessions provide the context for communication between Sources and Targets. A *Link Endpoint* associates a Terminus with a *Session Endpoint*. Within a Session, the Link Protocol (defined in the links section) is used to establish Links between Sources and Targets and to transfer Messages across them. A single Session may be simultaneously associated with 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)
```

3.2 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 (MAJOR), 0 (MINOR), 0 (REVISION)). 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.

```
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
```

Figure 3.2: Version Negotiation Examples

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

3.3 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.

3.3.1 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 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. (A type code of 0x01 indicates that the frame is an SASL frame, see sasl).

3.3.2 AMQP Frames

The AMQP frame type defines header bytes 6 and 7 to contain a Channel number (see the transport section). 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
            .                                   .  |
            .                                   .  |
            .                          _________|  |
            |                ...      |            |
            +--------------------------+          -'
```

An AMQP frame with no body may be used to generate artificial traffic as needed to satisfy any negotiated idle time-out interval. See doc-idle-time-out.

3.4 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
```

Although not strictly directed at the Connection endpoint, the `begin` and `end` Frames may be useful for the Connection endpoint to intercept as these Frames are how sessions mark the beginning and ending of communication on a given Channel.

3.4.1 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 (MIN-MAX-FRAME-SIZE) and the maximum channel number is 0. 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. The `open` frame can only be sent on channel 0. 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

                ...
```

3.4.2 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.

3.4.3 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. A `close` frame may be received on any channel up to the maximum channel number negotiated in open. However, implementations SHOULD send it on channel 0, and MUST send it on channel 0 if performing a pipelined open.

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

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

3.4.4 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
```

3.4.5 Idle Time Out Of A Connection

Connections are subject to an idle time-out threshold. The time-out is triggered by a local peer when no frames are received after a threshold value is exceeded. The idle time-out is measured in milliseconds, and starts from the time the last frame is received. If the threshold is exceeded, then a peer should try to gracefully close the connection using a `close` frame with an error explaining why. If the remote peer does not respond gracefully within a threshold to this, then the peer may close the TCP socket.

Each peer has its own (independent) idle time-out. At Connection open each peer communicates the maximum period between activity (frames) on the connection that it desires from its partner. The `open` frame carries the idle-time-out field for this purpose. To avoid spurious time-outs, the value in idle-time-out should be half the peer's actual timeout threshold.

If a peer can not, for any reason support a proposed idle time-out, then it should close the connection using a `close` frame with an error explaining why. There is no requirement for peers to support arbitrarily short or long idle time-outs.

The use of idle time-outs is any addition to any network protocol level control. Implementations should make use of TCP keep-alive wherever possible in order to be good citizens.

If a peer needs to satisfy the need to send traffic to prevent idle time-out, and has nothing to send, it may send an empty frame, i.e. a frame consisting solely of a frame header, with no frame body. This frame's channel can be any valid channel up to channel-max, but is otherwise to be ignored. Implementations SHOULD use channel 0 for empty frames, and MUST use channel 0 if channel-max has not yet been negotiated (i.e. before an `open` frame has been received). Apart from this use, empty frames have no meaning.

Empty frames can only be sent after the `open` frame is sent. As they are a frame, they should not be sent after the `close` frame has been sent.

As an alternative to using an empty frame to prevent an idle time-out, if a connection is in a permissible state, an implementation MAY choose to send a flow frame for a valid session.

If during operation a peer exceeds the remote peer's idle time-out's threshold, e.g. because it is heavily loaded, it SHOULD gracefully close the connection by using a `close` frame with an error explaining why.

3.4.6 Connection States

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

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

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

OPEN_PIPE
In this state we have sent both the Connection header and the `open` 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 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.

3.4.7 Connection State Diagram

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

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

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

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

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

3.5 Sessions

A Session is a bidirectional sequential conversation between two containers that provides a grouping for related links. Sessions serve as the context for link communication. Any number of links of any directionality can be *attached* to a given Session. However, a link may be attached to at most one Session at a time.

```
Link A-------+                          +------>Link A
             |                          |
            \|/         (attached)      |
Link B<--- Session <--------------> Session <---Link B

Link C------>*          (detached)      *------>Link C
```

Messages transferred on a link are sequentially identified within the Session. A session may be viewed as multiplexing link traffic, much like a connection multiplexes session traffic. However, unlike the sessions on a connection, links on a session are not entirely independent since they share a common sequence scoped to the session. This common sequence allows endpoints to efficiently refer to sets of message transfers regardless of the originating link. This is of particular benefit when a single application is receiving transfers along a large number of different links. In this case the session provides *aggregation* of otherwise independent links into a single stream that can be efficiently acknowledged by the receiving application.

3.5.1 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)
                                  /
                             <---+

                                ...

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

3.5.2 Ending A Session

Sessions end automatically when the Connection is closed or interrupted. Sessions are explicitly ended when either endpoint chooses to end the Session. When a Session is explicitly ended, an `end` frame is sent to announce the disassociation of the endpoint from its outgoing channel, and to carry error information when relevant.

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

                                          ...

      [CH3] END(error=...)        --------->                           (1)
                                         +-- [CH7] END(error=...)
                                        /
                                       /
  (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.
```

3.5.3 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(error=...)        --+   +-- [CH7] END(error=...)
  (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.
```

3.5.4 Session Errors

When a Session is unable to process input, it MUST indicate this by issuing an END with an appropriate `error` indicating the cause of the problem. 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(error=...)
                     \ /
                      x
                     / \
                 <--+   +--> *discarded*
END              ---------->
                     ...
=================================================
```

3.5.5 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.

DISCARDING
The DISCARDING state is a variant of the END_SENT state where the `end` is triggered by an error. In this case any incoming frames on the session MUST be silently discarded until the peer's `end` frame is received.

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

Figure 3.3: State Transitions

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.

3.5.6 Session Flow Control

The Session Endpoint assigns each outgoing `transfer` frame a *transfer-id* from a session scoped sequence. Each session endpoint maintains the following state to manage incoming and outgoing `transfer` frames:

next-incoming-id
The *next-incoming-id* identifies the expected transfer-id of the next incoming `transfer` frame.

incoming-window
The *incoming-window* defines the maximum number of incoming `transfer` frames that the endpoint can currently receive. This identifies a current maximum incoming transfer-id that can be computed by subtracting one from the sum of *incoming-window* and *next-incoming-id*.

next-outgoing-id
The *next-outgoing-id* is used to assign a unique sequence number to all outgoing transfer frames on a given session. The *next-outgoing-id* may be initialized to an arbitrary value and is incremented after each successive `transfer` according to RFC-1982 serial number arithmetic.

outgoing-window
The *outgoing-window* defines the maximum number of outgoing `transfer` frames that the endpoint can currently send. This identifies a current maximum outgoing transfer-id that can be computed by subtracting one from the sum of *outgoing-window* and *next-outgoing-id*.

remote-incoming-window
The *remote-incoming-window* reflects the maximum number of outgoing transfers that can be sent without exceeding the remote endpoint's incoming-window. This value MUST be decremented after every `transfer` frame is sent, and recomputed when informed of the remote session endpoint state.

remote-outgoing-window

The *remote-outgoing-window* reflects the maximum number of incoming transfers that may arrive without exceeding the remote endpoint's outgoing-window. This value MUST be decremented after every incoming `transfer` frame is received, and recomputed when informed fo the remote session endpoint state. When this window shrinks, it is an indication of outstanding transfers. Settling outstanding transfers may cause the window to grow.

Once initialized, this state is updated by various events that occur in the lifespan of a session and its associated links:

sending a transfer

Upon sending a transfer, the sending endpoint will increment its next-outgoing-id, decrement its remote-incoming-window, and may (depending on policy) decrement its outgoing-window.

receiving a transfer

Upon receiving a transfer, the receiving endpoint will update the next-incoming-id to match the transfer-id of the incoming transfer plus one, as well as decrementing the remote-outgoing-window, and may (depending on policy) decrement its incoming-window.

receiving a flow

When the endpoint receives a `flow` frame from its peer, it MUST update the *next-incoming-id* directly from the *next-outgoing-id* of the frame, as well as copy the *remote-outgoing-window* directly from the *outgoing-window* of the frame.

The *remote-incoming-window* is computed as follows:

$$\textit{next-incoming-id}_{\textit{flow}} + \textit{incoming-window}_{\textit{flow}} - \textit{next-outgoing-id}_{\textit{endpoint}}$$

If the *next-incoming-id* field of the `flow` frame is not set, then *remote-incoming-window* is computed as follows:

$$\textit{initial-outgoing-id}_{\textit{endpoint}} + \textit{incoming-window}_{\textit{flow}} - \textit{next-outgoing-id}_{\textit{endpoint}}$$

3.6 Links

A Link provides a unidirectional transport for Messages between a Source and a Target. The primary responsibility of a Source or Target (a Terminus) is to maintain a record of the status of each active delivery attempt until such a time as it is safe to forget. These are referred to as *unsettled* transfers. When a Terminus forgets the state associated with a delivery-tag, it is considered *settled*. Each delivery attempt is assigned a unique *delivery-tag* at the Source. The status of an active delivery attempt is known as the *Transfer State* of the delivery.

Link Endpoints interface between a Terminus and a Session Endpoint, and maintain additional state used for active communication between the local and remote endpoints. Link Endpoints therefore come in two flavors: *Senders* and *Receivers*. When the sending application submits a Message to the Sender for transport, it also supplies the delivery-tag used by the Source to track the Transfer State. The Link Endpoint assigns the Message data unique *transfer-ids* from a Session scoped sequence. These transfer-ids are used to efficiently reference subsets of the outstanding deliveries on a Session.

Termini may exist beyond their associated Link Endpoints, so it is possible for a Session to terminate and the Termini to remain. A Link is said to be *suspended* if the Termini exist, but have no associated Link Endpoints. The process of associating new Link Endpoints with existing Termini and reestablishing communication is referred to as *resuming* a Link.

The original Link Endpoint state is not necessary for resumption of a Link. Only the unsettled Transfer State maintained at the Termini is necessary for link resume, and this need not be stored directly.

The form of delivery-tags is intentionally left open-ended so that they and their related Transfer State can, if desired, be (re)constructed from application state, thereby minimizing or eliminating the need to retain additional protocol-specific state in order to resume a Link.

3.6.1 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.

3.6.2 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. The locally chosen handle is referred to as the *output handle*. The remotely chosen handle is referred to as the *input handle*.

At an Endpoint, a Link is considered to be *attached* when the Link Endpoint exists and has both input and output handles assigned at an active Session Endpoint. A Link is considered to be *detached* when the Link Endpoint exists, but is not assigned either input or output handles. A Link can be considered *half attached* (or *half detached*) when only one of the input or output handles is assigned.

```
+-------------------+                             +-------------------+
|    name: Link_1   |                             |    name: Link_1   |
|  handle: i        |                             |  handle: j        |
|-------------------|                             |-------------------|
|    role: receiver |<---+                   +--->|    role: sender   |
|  source: A        |    |                   |    |  source: A        |
|  target: B        |    |                   |    |  target: B        |
+-------------------+    |                   |    +-------------------+
                         |                   |
                         |    +---------+    |
        ...          <---+--->| Session |<---+--->        ...
                         |    +---------+    |
                         |                   |
+-------------------+    |                   |    +-------------------+
|    name: Link_N   |    |                   |    |    name: Link_N   |
|  handle: k        |<---+                   +--->|  handle: l        |
|-------------------|                             |-------------------|
|    role: sender   |                             |    role: receiver |
|  source: C        |                             |  source: C        |
|  target: D        |                             |  target: D        |
+-------------------+                             +-------------------+
```

3.6.3 Establishing Or Resuming A Link

Links are established and/or resumed by creating a Link Endpoint associated with a local Terminus, assigning it to an unused handle, and sending an `attach` Frame. This frame carries the state of the newly created Link Endpoint, including the local and remote termini, one being the source and one being the target depending on the directionality of the Link Endpoint. On receipt of the `attach`, the remote Session Endpoint creates a corresponding Link Endpoint and informs its application of the attaching Link. The application attempts to locate the Terminus previously associated with the Link. This Terminus is associated with the Link Endpoint and may be updated if its properties do not match those sent by the remote Link Endpoint. If no such Terminus exists, the application MAY choose to create one using the properties supplied by the remote Link Endpoint. The Link Endpoint is then mapped to an unused handle, and an `attach` Frame is issued carrying the state of the newly created endpoint. Note that if the application chooses not to create a Terminus, the Session Endpoint will still create a Link Endpoint and issue an `attach` indicating that the Link Endpoint has no associated local terminus. In this case, the Session Endpoint MUST immediately detach the newly created Link Endpoint.

```
Peer                                  Partner
================================================================
*create link endpoint*
ATTACH(name=N, handle=1,    ----------> *create link endpoint*
       role=sender,                +--- ATTACH(name=N, handle=2,
       source=A,                  /            role=receiver,
       target=B)                 /             source=A,
                                /              target=B)
                            <--+
                              ...
----------------------------------------------------------------
```

Figure 3.4: Establishing a Link

If there is no pre-existing Terminus, and the peer does not wish to create a new one, this is indicated by setting the local terminus (source or target as appropriate) to null.

```
    Peer                                  Partner
    ================================================================
    *create link endpoint*
    ATTACH(name=N, handle=1,    ----------> *create link endpoint*  (1)
           role=sender,                +--- ATTACH(name=N, handle=2,
           source=A,                  /            role=receiver,
           target=B)                 /             source=A,
                                    /              target=-)
(2)                             <--+
                                       +--- DETACH(handle=2,
                                      /            closed=True)
                                     /
                                    /
                                <--+
    DETACH(handle=1,            ----------->
           closed=True)
                                  ...
    ----------------------------------------------------------------
      (1) The Link Endpoint is created, but no target is created.
      (2) At this point the link is established, but it is to a
          nonexistent target.
```

Figure 3.5: Refusing a Link

If either end of the Link is already associated with a Terminus, the `attach` frame MUST include its unsettled transfer state.

```
    Peer                                       Partner
    ================================================================
    *existing source*
    ATTACH(name=N, handle=1,   ----------> *found existing target*
           role=sender,               +--- ATTACH(name=N, handle=2,  (1)
           source=X,                 /            role=receiver,
           target=Y,                /             source=X,
           unsettled=...)          /              target=Y,
(2)                            <--+               unsettled=...)
                                   ...
    ----------------------------------------------------------------
      (1) The target already exists, and its properties
          match the peer's expectations.
      (2) At this point the Link is reestablished with source=X,
          target=Y.
```

Figure 3.6: Resuming a Link

Note that the expected Terminus properties may not always match the actual Terminus properties reported by the remote endpoint. In this case, the Link is always considered to be between the Source as described by the Sender, and the Target as described by the Receiver. This can happen both when establishing and when resuming a link.

When a link is established, an endpoint may not have all the capabilities necessary to create the terminus exactly matching the expectations of the peer. Should this happen, the endpoint MAY adjust the properties in order to succeed in creating the terminus. In this case the endpoint MUST report the actual properties of the terminus as created.

When resuming a link, the Source and Target properties may have changed while the link was suspended. When this happens, the Termini properties communicated in the source and target fields of the `attach` frames may be in conflict. In this case, the Sender is considered to hold the authoritative version of the Source properties, the Receiver is considered to hold the authoritative version of the Target properties. As above, the resulting Link is constructed to be between the Source as described by the Sender, and the Target as described by the Receiver. Once the Link is resumed, either peer is free to continue if the updated properties are acceptable, or if not, `detach`.

```
    Peer                                     Partner
    ================================================================
    *existing source*
    ATTACH(name=N, handle=1,   ----------> *found existing target*
           role=sender,               +--- ATTACH(name=N, handle=2,  (1)
           source=A,                 /            role=receiver,
           target=B,                /             source=A,
           unsettled=...)          /              target=C,
(2)                            <--+               unsettled=...)
                                   ...
    ----------------------------------------------------------------
      (1) The Terminus already exists, but its state
          does not match the Peer's endpoint.
      (2) At this point the Link is established with source=A,
          target=C.
```

Figure 3.7: Resuming an altered Link

It is possible to resume a Link even if one of the Termini has lost nearly all its state. All that is required is the Link name and direction. This is referred to as *recovering* a Link. 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. If the remote peer has no record of the Link, then no terminus will be located, and local terminus (source or target as appropriate) field in the `attach` frame will be null.

```
    Peer                                       Partner
    ================================================================
    *create link endpoint*
    ATTACH(name=N, handle=1,   ----------> *found existing target*
           role=sender,              +--- ATTACH(name=N, handle=2,  (1)
           source=X                 /            role=receiver,
           target=-)               /             source=X,
(2)                            <---+             target=Y)
                                   ...
    ----------------------------------------------------------------
      (1) The target already exists, and its properties are
          authoritative.
      (2) At this point the Link is reestablished with source=X,
          target=Y.
```

Figure 3.8: Recovering a Link

3.6.4 Detaching And Reattaching A Link

A Session Endpoint can choose to unmap its output handle for a Link. In this case, the endpoint MUST send a `detach` frame to inform the remote peer that the handle is no longer attached to the Link Endpoint. Should both endpoints do this, the Link may return to a fully detached state. Note that in this case the Link Endpoints may still indirectly communicate via the Session, as there may be active transfers on the link referenced via transfer-id.

```
       Peer                                    Partner
    ================================================================
       *create link endpoint*
       ATTACH(name=N, handle=1   ----------> *create link endpoint*
              role=sender,             +--- ATTACH(name=N, handle=2,
              source=A,               /            role=receiver,
              target=B)              /             source=A,
                                    /              target=B)
                                 <--+
                                     ...
       *use link*                <---------> *use link*
                                     ...
       DETACH(handle=1)          ----------> *detach input handle*
   (1) *detach output handle*    <---------- DETACH(handle=2)
                                     ...
    ----------------------------------------------------------------
         (1) At this point both endpoints are detached.
```

When the state of a Link Endpoint changes, this is can be communicated by detaching and then reattaching with the updated state on the `attach` frame. This can be used to update the properties of the link endpoints, or to update the properties of the Termini.

```
    Peer                                Partner
    =============================================================
                                  ...
    DETACH(handle=1)          ---+
                                  \
                                   \
                                    \
    *modify link endpoint*           \
                                      +--> *detach input handle*
    ATTACH(name=N, handle=1   ---+    +--- DETACH(handle=2)
           role=sender,           \  /
           source=A’,              \/
           target=B’)              /\
                                  /  \
     *detach input handle*    <--+    +--> *reattach input handle*
                                           *modify link endpoint*
                                      +--- ATTACH(name=N, handle=2
                                     /            role=receiver,
                                    /             source=A’,
                                   /              target=B’)
                                  /
(1)  *reattach input handle*  <--+
                                  ...
    *use link*               <---------> *use link*
                                  ...
    -------------------------------------------------------------
      (1) At this point the link is updated and attached.
```

If an attempt to reattach a Link is made, but there is no existing Link Endpoint, then the attach request MUST be ignored, and subsequent communication along the input handle MUST be ignored until a subsequent detach occurs.

3.6.5 Link Errors

When an error occurs at a Link Endpoint, the endpoint MUST be detached with appropriate error information supplied in the error field of the `detach` frame. The Link Endpoint MUST then be destroyed. Should any input (other than a detach) related to the endpoint either via the input handle or transfer-ids be received, the session MUST be terminated with an errant-link `session-error`. Since the Link Endpoint has been destroyed, the peer cannot reattach, and MUST resume the link in order to restore communication.

3.6.6 Closing A Link

A peer closes a Link by sending the `detach` frame with the handle for the specified Link, and the closed flag set to true. The partner will destroy the corresponding Link endpoint, and reply with its own `detach` frame with the closed flag set to true.

```
    Peer                                Partner
    =============================================================
    *create link endpoint*
    ATTACH(name=N, handle=1   ----------> *create link endpoint*
           role=sender,              +--- ATTACH(name=N, handle=2,
           source=A,                /            role=receiver,
           target=B)               /             source=A,
                                  /              target=B)
                          <--+
                              ...
    *use link*            <---------> *use link*
                              ...
    DETACH(handle=1,      ----------> *destroy link endpoint*
           closed=True)
(1) *destroy link endpoint*  <---------- DETACH(handle=2,
                                                 closed=True)
    -------------------------------------------------------------
      (1) At this point both endpoints are destroyed.
```

Figure 3.9: Closing a Link

Note that one peer may send a closing detach while its partner is sending a non-closing detach. In this case, the partner MUST signal that it has closed the link by reattaching and then sending a closing detach.

3.6.7 Flow Control

Once attached, a Link is subject to flow control of Message transfers. Link Endpoints maintain the following flow control state. This state defines when it is legal to send transfers on an attached Link, as well as indicating when certain interesting conditions, such as insufficient transfers to consume the currently available *link-credit*, or insufficient *link-credit* to send available transfers:

transfer-count

The *transfer-count* is initialized by the Sender when a Link Endpoint is created, and is incremented whenever a Message is sent (at the Sender) or received (at the Receiver). Only the Sender may independently modify this field. The Receiver's value is calculated based on the last known value from the Sender and any subsequent Messages received on the Link.

link-credit

The *link-credit* variable defines the current maximum legal amount that the *transfer-count* may be increased. This identifies a *transfer-limit* that may be computed by adding the *link-credit* to the *transfer-count*.

Only the Receiver can independently choose a value for this field. The Sender's value MUST always be maintained in such a way as to match the *transfer-limit* identified by the Receiver. This means that the Sender's link-credit variable MUST be set according to this formula when flow information is given by the receiver:

$\textit{link-credit}_{\mathrm{snd}} := \textit{transfer-count}_{\mathrm{rcv}} + \textit{link-credit}_{\mathrm{rcv}}$ - $\textit{transfer-count}_{\mathrm{snd}}$.

In the event that the receiver does not yet know the *transfer-count*, i.e. $\textit{transfer-count}_{\mathrm{rcv}}$ is unspecified, the Sender MUST assume that the $\textit{transfer-count}_{\mathrm{rcv}}$ is the first $\textit{transfer-count}_{\mathrm{snd}}$ sent from Sender to Receiver, i.e. the $\textit{transfer-count}_{\mathrm{snd}}$ specified in the flow state carried by the initial `attach` frame from the Sender to the Receiver.

Additionally, whenever the Sender increases *transfer-count*, it MUST decrease *link-credit* by the same amount in order to maintain the *transfer-limit* identified by the Receiver.

available

The *available* variable is controlled by the Sender, and indicates to the Receiver, that the Sender could make use of the indicated amount of *link-credit*. Only the Sender can independently modify this field. The Receiver's value is calculated based on the last known value from the Sender and any subsequent incoming Messages received. The Sender MAY transfer Messages even if the available variable is zero. Should this happen, the Receiver MUST maintain a floor of zero in it's calculation of the value of available.

drain
The drain flag indicates how the Sender should behave when insufficient transfers are available to consume the current link-credit. If set, the Sender will (after sending all available transfers) advance the transfer-count as much as possible, consuming all link-credit, 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.

If the link-credit is less than or equal to zero, i.e. the transfer-count is the same as or greater than the transfer-limit, it is illegal to send more transfers. If the link-credit 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.

```
    +----------+                                +----------+
    |  Sender  |---------------transfer------------>| Receiver |
    +----------+                                +----------+
     \        / <----------------flow--------------- \        /
      +------+                                    +------+
          |
          |
          |
if link-credit <= 0 then pause
```

If the Sender's drain flag is set and there are no available transfers, the Sender MUST advance its transfer-count until link-credit is zero, and send its updated `flow` state to the Receiver.

The transfer-count is an absolute value. While the value itself is conceptually unbounded, it is encoded as a 32-bit integer that wraps around and compares according to RFC-1982 serial number arithmetic.

The initial flow state of a Link Endpoint is determined as follows. The *link-credit* and *available* variables are initialized to zero. The *drain* flag is initialized to False. The Sender may choose an arbitrary point to initialize the *transfer-count*. This value is communicated in the initial `attach` frame. The Receiver initializes its *transfer-count* upon receiving the Sender's `attach`.

```
                              flow state
                                   |
                                   | modifies
+-------------------+              |                +-------------------+
|      Sender       |    .-----------------------.  |     Receiver      |
+-------------------+     attach, transfer, flow    +-------------------+
| transfer-count    |-------------------------------->| transfer-count    |
| link-credit       |                               | link-credit       |
| available         |<--------------------------------| available         |
| drain             |             flow              | drain             |
+-------------------+            '-----'            +-------------------+
                                   |
                                   | modifies
                                   |
                              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 link-credit 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(link-credit=1, ...)      ---->
   timed-get: flow(link-credit=1, ...),
              *wait*,
              flow(drain=True, ...)          ---->
async-notify: flow(link-credit=delta, ...)  ---->
        stop: flow(link-credit=0, ...)       ---->
```

3.6.8 Synchronous Get

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

```
Receiver                                      Sender
=================================================================
                                       ...
flow(link-credit=1)              ---------->
                                         +---- transfer(...)
*block until transfer arrives*          /
                                 <---+
                                       ...
-----------------------------------------------------------------
```

Synchronous get with a timeout is accomplished by incrementing the link-credit, sending the updated `flow` state and waiting for the link-credit to be consumed. 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 link-credit to be consumed. Even if no transfers are available, this condition will be met promptly because of the drain flag. Once the link-credit is consumed, the Receiver can unambiguously determine whether a transfer has arrived or whether the operation has timed out.

```
    Receiver                                      Sender
  =================================================================
                                           ...
    flow(link-credit=1)              ---------->
  *wait for link-credit <= 0*
    flow(drain=True)                 ---+   +--- transfer(...)
                                         \ /
                                          x
                                         / \
(1)                                  <--+   +-->
(2)                                  <---------- flow(...)
                                           ...
  -----------------------------------------------------------------
      (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 until link-credit is consumed.
```

3.6.9 Asynchronous Notification

Asynchronous notification can be accomplished as follows. The receiver maintains a target amount of link-credit for that Link. As `transfer` arrive on the Link, the Sender's link-credit decreases as the transfer-count increases. When the Sender's link-credit falls below a threshold, the `flow` state may be sent to increase the Sender's link-credit back to the desired target.

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

3.6.10 Stopping A Link

Stopping the transfers on a given Link is accomplished by updating the link-credit to be zero 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
    =================================================================
                                           ...
    flow(...,                        <---------- transfer(...)
         link-credit=0,              ---+   +--- transfer(...)
         echo=True)                      \ /
                                          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.
```

3.6.11 Messages

The transport layer assumes as little as possible about Messages and allows alternative Message representations to be layered above. The transport defines Messages as a sequence of Sections. The number and meaning of Sections in a given Message is not defined at the transport layer. Sections may be used by layered Message representations to expose their internal structure to the transport if desired. Sections may also be used by a layered Message representation to encode boundaries that would otherwise require internal Message framing, e.g. demarcating a footer where the size of the body is not known up front.

```
  Section 1           Section 2            Section 3
+-----------+-------------------------------+-----------+
|  header   |             body              |  footer   |
+-----------+-------------------------------+-----------+
```

Figure 3.10: Example

3.6.11.1 Sections

Each Message Section has an associated format code that describes the data comprising that Section. The format code indicates the format and purpose of the section data. Format codes are carried by the transport along with the section data. Specific format codes are defined outside the transport.

Sections may be arbitrarily large, and Messages may have an arbitrarily large number of Sections.

3.6.11.2 Fragments

Messages are transferred as a list of fragments. Each fragment is a contiguous part of a **single** section, possibly being a whole section. Each fragment includes a first flag, last flag, and section-code code that 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 section-code fields could be used to indicate the boundary between the header and body, and to indicate the distinct formatting of each.

```
  Section 0           Section 1
+-----------+-------------------------------+
|  header   |              body             |
+-----------+-------+-----------+-----------+
|    F1     |  F2   |    F3     |    F4     |
+-----------+-------+-----------+-----------+
```

Fragment	first	last	section-code
F1	true	true	0xDB
F2	true	false	0xCA
F3	false	false	0xCA
F4	false	true	0xCA

Additionally, each `fragment` carries a *section-number* and *section-offset*. The former identifies which section the fragment belongs to, the latter indicates the offset of the fragment data within the section. These values aid in the efficient resumption of large messages where transfer has been interrupted by link detachment. Within a single delivery attempt for a Message, Sections (and their constituent fragments) MUST be sent in order.

3.6.12 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.

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

Figure 3.11: Initial Transfer

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.

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

Figure 3.12: Initial Receipt

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.

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

Figure 3.13: Indication of Non-Terminal State

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.

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

Figure 3.14: Indication of Terminal State

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.

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

Figure 3.15: Receipt of Terminal State

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.

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

Figure 3.16: Indication of Settlement

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.

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

Figure 3.17: Receipt of Settlement

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.

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

Figure 3.18: Final Settlement

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.

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

Figure 3.19: At-Most-Once

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.

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

Figure 3.20: At-Least-Once

As one might 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.

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.

3.6.13 Resuming Transfers

When a suspended link having unsettled transfers is resumed, the *unsettled* field from the `attach` frame will carry the delivery-tags and transfer-state of all transfers considered unsettled by the issuing link endpoint. The set of delivery tags and transfer-states contained in the unsettled maps from both endpoints can be divided into three categories:

Transfers that only the Source considers unsettled
Transfers in this category MAY be resumed at the discretion of the sending application.

Transfers that only the Target considers unsettled
Transfers in this category MUST be ignored by the Sender, and MUST be considered settled by the Receiver.

Transfers that both the Source and Target consider unsettled
Transfers in this category MUST be resumed by the Sender.

A transfer is resumed much the same way it is initially transferred with the following exceptions:

- The resume flag MUST be set to true when resuming a transfer.
- The Sender MAY omit message data when the Transfer State of the Receiver indicates retransmission is unnecessary.

Note that unsettled delivery-tags do NOT have any valid transfer-ids associated until they are resumed, as the transfer-ids from their original link endpoints are meaningless to the new link endpoints.

3.6.14 Transferring Large Messages

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. When a message is split up into multiple `transfer` frames in this manner, messages being transferred along different links MAY be interleaved. However, messages transferred along a single link MUST NOT be interleaved.

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 fragmented so long as the section boundaries and formatting are still recoverable from the first, last, and section-code fields.

```
              +------------+  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=?)
```

Figure 3.21: Outgoing 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=?)
```

Figure 3.22: Incoming Fragmentation State Diagram

3.7 Frame Bodies

3.7.1 Open

Negotiate Connection parameters.

```
<type name="open" class="composite" source="list" provides="frame">
    <descriptor name="amqp:open:list" code="0x00000000:0x00000010"/>
    <field name="options" type="options"/>
    <field name="container-id" type="string" mandatory="true"/>
    <field name="hostname" type="string"/>
    <field name="max-frame-size" type="uint" default="4294967295"/>
    <field name="channel-max" type="ushort" default="65535"/>
    <field name="idle-time-out" type="milliseconds"/>
    <field name="outgoing-locales" type="string" multiple="true"/>
    <field name="incoming-locales" type="string" multiple="true"/>
    <field name="offered-capabilities" type="symbol" multiple="true"/>
    <field name="desired-capabilities" type="symbol" multiple="true"/>
    <field name="properties" type="fields"/>
</type>
```

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 map*

`container-id` *the id of the source container*

`hostname` *the name of the target host*

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. This field can be used by AMQP proxies to determine the correct back-end service to connect the client to.
This field may already have been specified by the sasl-init frame, if a SASL layer is used, or, the server name indication extension as described in RFC-4366, if a TLS layer is used, in which case this field SHOULD be null or contain the same value. It is undefined what a different value to those already specific means.

`max-frame-size` *proposed maximum frame size*

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 (MIN-MAX-FRAME-SIZE) octets large.

`channel-max` *the maximum channel number that may be used on the Connection*

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.

`idle-time-out` *idle time-out*

The idle time-out required by the sender. A value of zero is the same as if it was not set (null). If the receiver is unable or unwilling to support the idle time-out then it should close the connection with an error explaining why (eg, because it is too small). If the value is not set, then the sender does not have an idle time-out. However, senders doing this should be aware that implementations MAY choose to use an internal default to efficiently manage a peer's resources.

`outgoing-locales` *locales available for outgoing text*

A list of the locales that the peer supports for sending informational text. This includes Connection, Session and Link error descriptions. 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. A null value or an empty list implies that only en_US is supported.

`incoming-locales` *desired locales for incoming text in decreasing level of preference*

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. A null value or an empty list implies that only en_US is supported.

`offered-capabilities` *the extension capabilities the sender supports*

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` *the extension capabilities the sender may use if the receiver supports them*

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.

`properties` *connection properties*

The properties map contains a set of fields intended to indicate information about the connection and its container.

3.7.2 Begin

Begin a Session on a channel.

```
<type name="begin" class="composite" source="list" provides="frame">
    <descriptor name="amqp:begin:list" code="0x00000000:0x00000011"/>
    <field name="options" type="options"/>
    <field name="remote-channel" type="ushort"/>
    <field name="next-outgoing-id" type="transfer-number" mandatory="true"/>
    <field name="incoming-window" type="uint" mandatory="true"/>
    <field name="outgoing-window" type="uint" mandatory="true"/>
    <field name="handle-max" type="handle" default="4294967295"/>
    <field name="offered-capabilities" type="symbol" multiple="true"/>
    <field name="desired-capabilities" type="symbol" multiple="true"/>
    <field name="properties" type="fields"/>
</type>
```

Indicate that a Session as begun on the channel.

Field Details

`options`	*options map*	
`remote-channel`	*the remote channel for this Session*	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.
`next-outgoing-id`		
`incoming-window`		
`outgoing-window`		
`handle-max`	*the maximum handle value that may be used on the Session*	The handle-max value is the highest handle value that may be used on the Session. A peer MUST NOT attempt to attach a Link using a handle value outside the range that its partner can handle. A peer that receives a handle outside the supported range MUST close the Connection with the framing-error error-code.
`offered-capabilities`	*the extension capabilities the sender supports*	
`desired-capabilities`	*the extension capabilities the sender may use if the receiver supports them*	
`properties`	*session properties*	The properties map contains a set of fields intended to indicate information about the session and its container.

3.7.3 Attach

Attach a Link to a Session.

```
<type name="attach" class="composite" source="list" provides="frame">
    <descriptor name="amqp:attach:list" code="0x00000000:0x00000012"/>
    <field name="options" type="options"/>
    <field name="name" type="string" mandatory="true"/>
    <field name="handle" type="handle" mandatory="true">
        <error name="handle-busy" type="amqp-error" value="invalid-field"/>
    </field>
    <field name="role" type="role" mandatory="true"/>
    <field name="source" type="*" requires="source"/>
    <field name="target" type="*" requires="target"/>
    <field name="unsettled" type="map"/>
    <field name="initial-transfer-count" type="sequence-no"/>
    <field name="max-message-size" type="ulong"/>
    <field name="offered-capabilities" type="symbol" multiple="true"/>
    <field name="desired-capabilities" type="symbol" multiple="true"/>
    <field name="properties" type="fields"/>
</type>
```

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.

Field Details

`options` *options map*

`name` *the name of the link*

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*.

`handle`

handle-busy error: invalid-field
The handle MUST NOT be used for other open Links.

`role` *role of the link endpoint*

`source` *the source for Messages*

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*

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.

`unsettled` *unsettled transfer state*

This is used to indicate any unsettled transfer states when a suspended link is resumed. 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. See resuming-transfers.
When reattaching, the unsettled map MUST be null.

`initial-transfer-count`

This MUST NOT be null if role is sender, and it is ignored if the role is receiver.

`max-message-size` *the maximum message size supported by the link endpoint*

This field indicates the maximum message size supported by the link endpoint. Any attempt to transfer a message larger than this results in a message-size-exceeded `link-error`. If this field is zero or unset, there is no maximum size imposed by the link endpoint.

`offered-capabilities` *the extension capabilities the sender supports*

`desired-capabilities` *the extension capabilities the sender may use if the receiver supports them*

`properties` *link properties*

The properties map contains a set of fields intended to indicate information about the link and its container.

3.7.4 Flow

Update link state.

```
<type name="flow" class="composite" source="list" provides="frame">
    <descriptor name="amqp:flow:list" code="0x00000000:0x00000013"/>
    <field name="options" type="options"/>
    <field name="next-incoming-id" type="transfer-number"/>
    <field name="incoming-window" type="uint" mandatory="true"/>
    <field name="next-outgoing-id" type="transfer-number" mandatory="true"/>
    <field name="outgoing-window" type="uint" mandatory="true"/>
    <field name="handle" type="handle">
        <error name="unattached-handle" type="amqp-error" value="invalid-field"/>
    </field>
    <field name="transfer-count" type="sequence-no"/>
    <field name="link-credit" type="uint"/>
    <field name="available" type="uint"/>
    <field name="drain" type="boolean" default="false"/>
    <field name="echo" type="boolean" default="false"/>
</type>
```

Updates the flow state for the specified Link.

Field Details

`options` *options map*

`next-incoming-id`

Identifies the expected transfer-id of the next incoming `transfer` frame. This value is not set if and only if the sender has not yet received the `begin` frame for the session. See session-flow-control for more details.

`incoming-window`

Defines the maximum number of incoming `transfer` frames that the endpoint can currently receive. See session-flow-control for more details.

`next-outgoing-id`

The transfer-id that will be assigned to the next outgoing `transfer` frame. See session-flow-control for more details.

`outgoing-window`

Defines the maximum number of outgoing `transfer` frames that the endpoint could potentially currently send, if it was not constrained by restrictions imposed by its peer's incoming-window. See session-flow-control for more details.

`handle`

unattached-handle error: invalid-field

If set, indicates that the flow frame carries flow state information for the local Link Endpoint associated with the given handle. If not set, the flow frame is carrying only information pertaining to the Session Endpoint.

`transfer-count` *the endpoint's transfer-count*

When the handle field is not set, this field MUST NOT be set.
When the handle identifies that the flow state is being sent from the Sender Link Endpoint to Receiver Link Endpoint this field MUST be set to the current transfer-count of the Link Endpoint.
When the flow state is being sent from the Receiver Endpoint to the Sender Endpoint this field MUST be set to the last known value of the corresponding Sending Endpoint. In the event that the Receiving Link Endpoint has not yet seen the initial `attach` frame from the Sender this field MUST NOT be set.
See flow-control for more details.

`link-credit` *the current maximum number of Messages that can be received*

The current maximum number of Messages that can be handled at the Receiver Endpoint of the Link. Only the receiver endpoint can independently set this value. The sender endpoint sets this to the last known value seen from the receiver. See flow-control for more details.
When the handle field is not set, this field MUST NOT be set.

`available` *the number of available Messages*

The number of Messages awaiting credit at the link sender endpoint. Only the sender can independently set this value. The receiver sets this to the last known value seen from the sender. See flow-control for more details.
When the handle field is not set, this field MUST NOT be set.

`drain` *indicates drain mode*

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. See flow-control for more details.
When the handle field is not set, this field MUST NOT be set.

`echo` *request link state from other endpoint*

3.7.5 Transfer

Transfer a Message.

```
<type name="transfer" class="composite" source="list" provides="frame">
    <descriptor name="amqp:transfer:list" code="0x00000000:0x00000014"/>
    <field name="options" type="options"/>
    <field name="handle" type="handle" mandatory="true">
        <error name="unattached-handle" type="amqp-error" value="invalid-field"/>
    </field>
    <field name="delivery-tag" type="delivery-tag" mandatory="true"/>
    <field name="transfer-id" type="transfer-number" mandatory="true"/>
    <field name="settled" type="boolean" default="false"/>
    <field name="state" type="*" requires="transfer-state"/>
    <field name="resume" type="boolean" default="false"/>
    <field name="more" type="boolean" default="false"/>
    <field name="aborted" type="boolean" default="false"/>
    <field name="batchable" type="boolean" default="false"/>
    <field name="message-format" type="message-format" mandatory="true"/>
    <field name="fragments" type="fragment" multiple="true"/>
</type>
```

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 map*

`handle`

Specifies the Link on which the Message is transferred.
unattached-handle error: invalid-field

`delivery-tag`

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

`transfer-id` *alias for delivery-tag*

`settled`

`state`

`resume` *indicates a resumed transfer*

If true, the resume flag indicates that the transfer is being used to reassociate an unsettled transfer from a dissociated link endpoint. See resuming-transfers for more details.

`more` *indicates that the Message has more content*

`aborted` *indicates that the Message is aborted*

Aborted Messages should be discarded by the recipient.

`batchable` *batchable hint*

If true, then the issuer is hinting that there is no need for the peer to urgently communicate updated transfer state. This hint may be used to artificially increase the amount of batching an implementation uses when communicating transfer states, and thereby save bandwidth.

`message-format` *indicates the message format*

`fragments`

3.7.6 Disposition

Inform remote peer of transfer state changes.

```
<type name="disposition" class="composite" source="list" provides="frame">
    <descriptor name="amqp:disposition:list" code="0x00000000:0x00000015"/>
    <field name="options" type="options"/>
    <field name="role" type="role" mandatory="true"/>
    <field name="batchable" type="boolean" default="false"/>
    <field name="first" type="transfer-number" mandatory="true"/>
    <field name="last" type="transfer-number"/>
    <field name="settled" type="boolean" default="false"/>
    <field name="state" type="*" requires="transfer-state"/>
</type>
```

The disposition frame is used to inform the remote peer of local changes in the state of message transfers. The disposition frame may reference transfers from many different links associated with a session, although all links MUST have the directionality indicated by the specified *role*.

Field Details

`options`

`role` *directionality of disposition*

The role identifies whether the disposition frame contains information about *sending* link endpoints or *receiving* link endpoints.

`batchable` *batchable hint*

If true, then the issuer is hinting that there is no need for the peer to urgently communicate the impact of the updated transfer states. This hint may be used to artificially increase the amount of batching an implementation uses when communicating transfer states, and thereby save bandwidth.

`first` *lower bound of transfers*

Identifies the lower bound of transfer-ids for the transfers in this set.

`last` *upper bound of transfers*

Identifies the upper bound of transfer-ids for the transfers in this set. If not set, this is taken to be the same as *first*.

`settled` *indicates transfers are settled*

If true, indicates that the referenced transfers are considered settled by the issuing endpoint.

`state` *indicates state of transfers*

Communicates the transfer-state of all the transfers referenced by this disposition.

3.7.7 Detach

Detach the Link Endpoint from the Session.

```
<type name="detach" class="composite" source="list" provides="frame">
    <descriptor name="amqp:detach:list" code="0x00000000:0x00000016"/>
    <field name="options" type="options"/>
    <field name="handle" type="handle" mandatory="true"/>
    <field name="closed" type="boolean" default="false"/>
    <field name="error" type="error"/>
</type>
```

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 map*

`handle`

`closed`

`error` *error causing the detach*

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

3.7.8 End

End the Session.

```
<type name="end" class="composite" source="list" provides="frame">
    <descriptor name="amqp:end:list" code="0x00000000:0x00000017"/>
    <field name="options" type="options"/>
    <field name="error" type="error"/>
</type>
```

Indicates that the Session has ended.

Field Details

`options` *options map*

`error` *error causing the end*

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

3.7.9 Close

Signal a Connection close.

```
<type name="close" class="composite" source="list" provides="frame">
    <descriptor name="amqp:close:list" code="0x00000000:0x00000018"/>
    <field name="options" type="options"/>
    <field name="error" type="error"/>
</type>
```

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 map*

`error` *error causing the close*

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

3.8 Definitions

3.8.1 Role

Link endpoint role.

```
<type name="role" class="restricted" source="boolean">
    <choice name="sender" value="false"/>
    <choice name="receiver" value="true"/>
</type>
```

Valid Values

false

true

3.8.2 Handle

The handle of a Link.

```
<type name="handle" class="restricted" source="uint"/>
```

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.

3.8.3 Seconds

A duration measured in seconds.

```
<type name="seconds" class="restricted" source="uint"/>
```

3.8.4 Milliseconds

A duration measured in milliseconds.

```
<type name="milliseconds" class="restricted" source="uint"/>
```

3.8.5 Delivery Tag

```
<type name="delivery-tag" class="restricted" source="binary"/>
```

3.8.6 Transfer Number

```
<type name="transfer-number" class="restricted" source="sequence-no"/>
```

3.8.7 Sequence No

32-bit RFC-1982 serial number.

```
<type name="sequence-no" class="restricted" source="uint"/>
```

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

3.8.8 Message Format

32-bit message format code.

```
<type name="message-format" class="restricted" source="uint"/>
```

The upper three octets of a message format code identify a particular message format. The lowest octet indicates the version of said message format. Any given version of a format is forwards compatible with all higher versions.

```
     3 octets      1 octet
 +----------------+---------+
 | message format | version |
 +----------------+---------+
 |                          |
msb                        lsb
```

3.8.9 Fragment

A Message fragment.

```
<type name="fragment" class="composite" source="list">
    <descriptor name="amqp:fragment:list" code="0x00000000:0x0000001c"/>
    <field name="first" type="boolean" mandatory="true"/>
    <field name="last" type="boolean" mandatory="true"/>
    <field name="section-code" type="uint" mandatory="true"/>
    <field name="section-number" type="uint" mandatory="true"/>
    <field name="section-offset" type="ulong" default="0"/>
    <field name="payload" type="binary"/>
</type>
```

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` *indicates the fragment is the first in the Section*

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

`last` *indicates the fragment is the last in the Section*

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

`section-code` *indicates the format of the Message section*

The section code indicates the format of the current Section of the Message. A Message may have multiple sections, and therefore multiple section codes, however the section code is only permitted to change at section boundaries (i.e. all fragments of the same Section MUST have the same section-code).

`section-number` *the current section number, the first section having section-number 0*

`section-offset` *the offset of the payload within the specified message section*

`payload` *Message data*

3.8.10 Fields

A mapping from field name to value.

```
<type name="fields" class="restricted" source="map"/>
```

The *fields* type is a map where the keys are restricted to be of type symbol. There is no further restriction implied by the *fields* type on the allowed values for the entries or the set of allowed keys.

3.8.11 Options

Options map.

```
<type name="options" class="restricted" source="fields"/>
```

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.

3.8.12 Error

Details of an error.

```
<type name="error" class="composite" source="list">
    <descriptor name="amqp:error:list" code="0x00000000:0x0000001d"/>
    <field name="condition" type="symbol" requires="error" mandatory="true"/>
    <field name="description" type="string"/>
    <field name="info" type="map"/>
</type>
```

Field Details

`condition` *error condition*

A symbolic value indicating the error condition.

`description` *descriptive text about the error condition*

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

`info` *map carrying information about the error condition*

3.8.13 Amqp Error

Shared error conditions.

```
<type name="amqp-error" class="restricted" source="symbol" provides="error">
    <choice name="internal-error" value="amqp:internal-error"/>
    <choice name="not-found" value="amqp:not-found"/>
    <choice name="unauthorized-access" value="amqp:unauthorized-access"/>
    <choice name="decode-error" value="amqp:decode-error"/>
    <choice name="resource-limit-exceeded" value="amqp:resource-limit-exceeded"/>
    <choice name="not-allowed" value="amqp:not-allowed"/>
    <choice name="invalid-field" value="amqp:invalid-field"/>
    <choice name="not-implemented" value="amqp:not-implemented"/>
    <choice name="resource-locked" value="amqp:resource-locked"/>
    <choice name="precondition-failed" value="amqp:precondition-failed"/>
    <choice name="resource-deleted" value="amqp:resource-deleted"/>
    <choice name="illegal-state" value="amqp:illegal-state"/>
</type>
```

Valid Values

amqp:internal-error

An internal error ocurred. Operator intervention may be required to resume normal operation.

amqp:not-found

A peer attempted to work with a remote entity that does not exist.

amqp:unauthorized-access

A peer attempted to work with a remote entity to which it has no access due to security settings.

amqp:decode-error

Data could not be decoded.

amqp:resource-limit-exceeded

A peer exceeded its resource allocation.

amqp:not-allowed

The peer tried to use a frame in a manner that is inconsistent with the semantics defined in the specification.

amqp:invalid-field

An invalid field was passed in a frame body, and the operation could not proceed.

amqp:not-implemented

The peer tried to use functionality that is not implemented in its partner.

amqp:resource-locked

The client attempted to work with a server entity to which it has no access because another client is working with it.

amqp:precondition-failed

The client made a request that was not allowed because some precondition failed.

amqp:resource-deleted

A server entity the client is working with has been deleted.

amqp:illegal-state

The peer sent a frame that is not permitted in the current state of the Session.

3.8.14 Connection Error

Symbols used to indicate connection error conditions.

```
<type name="connection-error" class="restricted" source="symbol" provides="error">
    <choice name="connection-forced" value="amqp:connection:forced"/>
    <choice name="framing-error" value="amqp:connection:framing-error"/>
</type>
```

Valid Values

amqp:connection:forced

An operator intervened to close the Connection for some reason. The client may retry at some later date.

amqp:connection:framing-error

A valid frame header cannot be formed from the incoming byte stream.

3.8.15 Session Error

Symbols used to indicate session error conditions.

```
<type name="session-error" class="restricted" source="symbol" provides="error">
    <choice name="window-violation" value="amqp:session:window-violation"/>
    <choice name="errant-link" value="amqp:session:errant-link"/>
</type>
```

Valid Values

amqp:session:window-violation

The peer violated incoming window for the session.

amqp:session:errant-link

Input was received for a link that was detached with an error.

3.8.16 Link Error

Symbols used to indicate link error conditions.

```
<type name="link-error" class="restricted" source="symbol" provides="error">
    <choice name="detach-forced" value="amqp:link:detach-forced"/>
    <choice name="transfer-limit-exceeded" value="amqp:link:transfer-limit-exceeded"/>
    <choice name="message-size-exceeded" value="amqp:link:message-size-exceeded"/>
</type>
```

Valid Values

amqp:link:detach-forced

An operator intervened to detach for some reason.

amqp:link:transfer-limit-exceeded
The peer sent more Message transfers than currently allowed on the session.

amqp:link:message-size-exceeded
The peer sent a larger message than is supported on the link.

3.8.17 PORT

PORT: the IANA assigned port number for AMQP 5672

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.

3.8.18 SECURE-PORT

SECURE-PORT: the IANA assigned port number for secure AMQP (amqps) 5671

The standard AMQP port number that has been assigned by IANA for secure TCP using TLS.

Implementations listening on this port should NOT expect a protocol handshake before TLS is negotiated.

3.8.19 MAJOR

MAJOR: major protocol version 1

3.8.20 MINOR

MINOR: minor protocol version 0

3.8.21 REVISION

REVISION: protocol revision 0

3.8.22 MIN-MAX-FRAME-SIZE

MIN-MAX-FRAME-SIZE: the minimum size (in bytes) of the maximum frame size	4096

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 4

Messaging

4.1 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
- distribution nodes
  - states for messages stored at a distribution node
- sources and targets
  - default disposition of transfers
  - supported outcomes
  - filtering of messages from a node
  - distribution-mode for access to messages stored at a distribution node
  - on-demand node creation

4.2 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.

```
                     Bare Message
                          |
                .----------+----------.
                |                     |
  +--------+ +------------+------+ +--------+
  | header | | properties | body | | footer |
  +--------+ +------------+------+ +--------+
 |                                           |
 '-------------------+----------------------'
                     |
               Annotated Message
```

The bare message is immutable within the AMQP network. That is neither the opaque body, nor the properties can be changed by any node acting as an AMQP intermediary.

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.

Altogether the message consists of the following sections:

1. Zero or one `header` sections for annotations at the head of the message.
2. Zero or one `properties` sections for standard properties in the bare message.
3. Zero or one application data sections of type amqp-map. Where the first section of the application data is an amqp-map the section is taken to contain "application properties" which may be used for purposes such as routing and filtering.
4. Zero or more application data sections (of any valid section type).
5. Zero or one `footer` sections 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 section-code as per the Message Fragmentation definition in the links section of Book III. The section codes are assigned according to `section-codes`.

4.2.1 Section Codes

```
<type name="section-codes" class="restricted" source="uint">
```

```
    <choice name="header" value="0"/>
    <choice name="properties" value="1"/>
    <choice name="footer" value="2"/>
    <choice name="data" value="3"/>
    <choice name="amqp-data" value="4"/>
    <choice name="amqp-map" value="5"/>
    <choice name="amqp-list" value="6"/>
</type>
```

Valid Values

0

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

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

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

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).

4

Section-code indicating an application data section. Sections of this type MUST consist of zero or more encoded AMQP values.

5

Section-code indicating an application data section. Sections of this type MUST be of zero length or consist of a single encoded instance of an AMQP `map`.

6

Section-code indicating an application data section. Sections of this type MUST be of zero length or consist of a single encoded instance of an AMQP `list`.

4.2.2 Header

Transport headers for a Message.

```
<type name="header" class="composite" source="list" provides="section">
    <descriptor name="amqp:header:list" code="0x00000000:0x00000020"/>
    <field name="durable" type="boolean"/>
    <field name="priority" type="ubyte"/>
    <field name="transmit-time" type="timestamp"/>
    <field name="ttl" type="ulong"/>
    <field name="former-acquirers" type="uint"/>
    <field name="delivery-failures" type="uint"/>
    <field name="message-attrs" type="message-attributes"/>
    <field name="delivery-attrs" type="message-attributes"/>
</type>
```

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

Field Details

`durable` *specify durability requirements*

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

`priority` *relative Message priority*

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` *the time of Message transmit*

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` *time to live in ms*

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`

The number of other Links that have acquired but failed to process the Message as indicated by the rejected or modified outcome (see the delivery-failed flag). This does not include the current Link even if delivery to the current Link has been previously attempted.

`delivery-failures` *the number of prior unsuccessful delivery attempts*

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 upon an outcome being settled at the sender, according to rules defined for each outcome.

`message-attrs` *message attributes*

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` *delivery attributes*

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.

4.2.3 Properties

Immutable properties of the Message.

```
<type name="properties" class="composite" source="list" provides="section">
    <descriptor name="amqp:properties:list" code="0x00000000:0x00000021"/>
    <field name="message-id" type="binary"/>
    <field name="user-id" type="binary"/>
    <field name="to" type="*" requires="address"/>
    <field name="subject" type="string"/>
    <field name="reply-to" type="*" requires="address"/>
    <field name="correlation-id" type="binary"/>
    <field name="content-length" type="ulong"/>
    <field name="content-type" type="symbol"/>
</type>
```

Message properties carry information about the Message.

Field Details

`message-id` *application Message identifier*

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` *creating user id*

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

`to` *the address of the Node the Message is destined for*

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` *the subject of the message*

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

`reply-to` *the Node to send replies to*

The address of the Node to send replies to.

`correlation-id` *application correlation identifier*

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

`content-length` *length of the combined payload in bytes*

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

`content-type` *MIME content type*

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”’.

4.2.4 Footer

Transport footers for a Message.

```
<type name="footer" class="composite" source="list" provides="section">
    <descriptor name="amqp:footer:list" code="0x00000000:0x00000022"/>
    <field name="message-attrs" type="message-attributes"/>
    <field name="delivery-attrs" type="message-attributes"/>
</type>
```

Field Details

`message-attrs` *message attributes*

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` *delivery attributes*

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.

4.2.5 Message Attributes

Message annotations.

`<type name="message-attributes" class="restricted" source="map"/>`

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.

4.3 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

4.3.1 Transfer State

The state of a message transfer.

```
<type name="transfer-state" class="composite" source="list" provides="transfer-state">
    <descriptor name="amqp:transfer-state:list" code="0x00000000:0x00000023"/>
    <field name="options" type="options"/>
    <field name="bytes-transferred" type="ulong"/>
    <field name="outcome" type="*" requires="outcome"/>
    <field name="txn-id" type="*" requires="txn-id"/>
</type>
```

Field Details

`options`	*options map*
`bytes-transferred`	
`outcome`	
`txn-id`	

4.3.2 Accepted

The accepted outcome.

```
<type name="accepted" class="composite" source="list" provides="outcome">
    <descriptor name="amqp:accepted:list" code="0x00000000:0x00000024"/>
    <field name="options" type="options"/>
</type>
```

The accepted outcome is used to indicate that an incoming Message has been successfully processed. The accepted outcome does not increment the *delivery-failures* count in the header of the accepted Message.

Field Details

`options`	*options map*

4.3.3 Rejected

The rejected outcome.

```
<type name="rejected" class="composite" source="list" provides="outcome">
    <descriptor name="amqp:rejected:list" code="0x00000000:0x00000025"/>
    <field name="options" type="options"/>
    <field name="error" type="error"/>
</type>
```

The rejected outcome is used to indicate that an incoming Message is invalid and therefore unprocessable. The rejected outcome when applied to a Message will cause the *delivery-failures* count to be incremented in the header of the rejected Message.

Field Details

`options`	*options map*
`error`	*error that caused the message to be rejected*

The value supplied in this field will be placed in the header of the rejected Message in the message-attrs map under the key "rejected".

4.3.4 Released

The released outcome.

```
<type name="released" class="composite" source="list" provides="outcome">
    <descriptor name="amqp:released:list" code="0x00000000:0x00000026"/>
    <field name="options" type="options"/>
</type>
```

The released outcome is used to indicate that a given transfer was not and will not be acted upon. An outcome of released MUST NOT cause the *delivery-failures* count of the header of the released Message to be incremented.

Field Details

`options`	*options map*

4.3.5 Modified

The modified outcome.

```
<type name="modified" class="composite" source="list" provides="outcome">
    <descriptor name="amqp:modified:list" code="0x00000000:0x00000027"/>
    <field name="options" type="options"/>
    <field name="delivery-failed" type="boolean"/>
    <field name="deliver-elsewhere" type="boolean"/>
    <field name="message-attrs" type="message-attributes"/>
    <field name="delivery-attrs" type="message-attributes"/>
</type>
```

Field Details

`options`	*options map*
`delivery-failed`	*count the transfer as an unsuccessful delivery attempt*

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

`deliver-elsewhere`	*prevent redelivery*

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

`message-attrs` *message attributes*

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` *delivery attributes*

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.

4.4 Distribution Nodes

4.4.1 Message States

The Messaging layer defines a set of states for Messages stored at a *distribution node*. Not all Nodes store Messages for distribution, however these definitions permit some standardized interaction with those nodes that do. The transitions between these states are controlled by the transfer of Messages to/from a distribution node and the resulting terminal transfer state. Note that the state of a Message at one distribution 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 an *acquiring* transfer, the Message will transition to the ACQUIRED state. Once in the ACQUIRED state, a Messages is ineligible for *acquiring* transfers to any other Links.

A Message will remain ACQUIRED at the distribution 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 *default-outcome* of the source is used.

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

```
                                  +------------+
                               +->| AVAILABLE  |
                               |  +------------+
                               |         |
                               |         |
       settled with outcome:   |         |
         RELEASED/MODIFIED     |         | TRANSFER (acquiring)
                               |         |
                               |         |
                               |        \|/
                               |  +------------+
                               +--|  ACQUIRED  |
                                  +------------+
                                         |
                                         |
                                         | settled with outcome:
                                         |   ACCEPTED/REJECTED
                                         |
                                         |
                                        \|/
                                  +------------+
                                  |  ARCHIVED  |
                                  +------------+
```

Figure 4.1: Message State Transitions

4.5 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.

4.5.1 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).

4.5.2 Distribution Modes

The Source defines an optional distribution-mode that informs and/or indicates how distribution nodes are to behave with respect to the Link. The distribution-mode of a Source determines how Messages from a distribution node are distributed among its associated Links. There are two defined distribution-modes: *move* and *copy*. When specified, the distribution-mode has two related effects on the behaviour of a distribution node with respect to the Link associated with the Source.

The *move* distribution-mode causes messages transferred from the distribution node to transition to the ACQUIRED state prior to transfer over the link, and subsequently to the ARCHIVED state when the transfer is settled with a successful outcome. The *copy* distribution-mode leaves the state of the Message unchanged at the distribution node.

A 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 *move* 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 *copy* 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.

4.5.3 Source

```
<type name="source" class="composite" source="list" provides="source">
    <descriptor name="amqp:source:list" code="0x00000000:0x00000028"/>
    <field name="options" type="options"/>
    <field name="address" type="*" requires="address">
        <error name="address-not-found" type="amqp-error" value="not-found"/>
    </field>
    <field name="durable" type="boolean"/>
    <field name="expiry-policy" type="terminus-expiry-policy" default="session"/>
    <field name="timeout" type="seconds" default="0"/>
    <field name="dynamic" type="*" requires="lifetime-policy"/>
    <field name="distribution-mode" type="symbol" requires="distribution-mode"/>
    <field name="filter" type="filter-set"/>
    <field name="default-outcome" type="*" requires="outcome"/>
    <field name="outcomes" type="symbol" multiple="true"/>
    <field name="capabilities" type="symbol" multiple="true"/>
</type>
```

Field Details

`options`	*options map*
`address`	*the address of the source*

address-not-found error: not-found

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.

`durable`	*indicates that the Source will be durably retained*
`expiry-policy`	*the expiry policy of the Source*
`timeout`	*duration that an expiring Source will be retained*

The Source starts expiring as indicated by the expiry-policy.

`dynamic`	*request dynamic creation of a remote Node*

When set in the local Linkage by the receiving Link endpoint, this field constitutes a request for the sending peer to dynamically create a Node at the source. In this case the address field MUST NOT be set.
When set in the local Linkage by the sending Link Endpoint this field indicates the lifetime policy in use for the Node which has been dynamically created at the time of the attach. In this case the address field will contain the address of the created Node. The generated address SHOULD include the Link name and Session-name or client-id in some recognizable form for ease of traceability.
The lifetime of the generated node is controlled by the policy specified in the dynamic field in the Source of the local Linkage of the sending Link Endpoint. Definitionally, the lifetime will never be less than the lifetime of the link which caused its creation.

`distribution-mode` *the distribution mode of the Link*

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` *a set of predicates to filter the Messages admitted onto the Link*

`default-outcome` *default outcome for unsettled transfers*

Indicates the outcome to be used for transfers that have not reached a terminal state at the receiver when the transfer is settled, including when the Link Endpoint is destroyed. The value MUST be a valid outcome (e.g. `released`, or `rejected`).

`outcomes` *descriptors for the outcomes that can be chosen on this link*

The values in this field are the symbolic descriptors of the outcomes that can be chosen on this link. This field MAY be empty, indicating that the *default-outcome* will be assumed for all message transfers. When present, the values MUST be a symbolic descriptor of a valid outcome, e.g. ”amqp:accepted:list”.

`capabilities` *the extension capabilities the sender supports/desires*

4.5.4 Target

```
<type name="target" class="composite" source="list" provides="target">
    <descriptor name="amqp:target:list" code="0x00000000:0x00000029"/>
    <field name="options" type="options"/>
    <field name="address" type="*" requires="address">
        <error name="address-not-found" type="amqp-error" value="not-found"/>
    </field>
    <field name="durable" type="boolean"/>
    <field name="expiry-policy" type="terminus-expiry-policy" default="session"/>
    <field name="timeout" type="seconds" default="0"/>
    <field name="dynamic" type="*" requires="lifetime-policy"/>
    <field name="capabilities" type="symbol" multiple="true"/>
</type>
```

Field Details

`options` *options map*

`address` *The address of the target.*

address-not-found error: not-found

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.

`durable` *indicates that the Target will be durably retained*

`expiry-policy` *the expiry policy of the Target*

`timeout` *duration that an expiring Target will be retained*

The Target starts expiring as indicated by the expiry-policy.

`dynamic` *request dynamic creation of a remote Node*

When set in the local Linkage by the sending Link endpoint, this field constitutes a request for the receiving peer to dynamically create a Node at the target. In this case the address field MUST NOT be set.
When set in the local Linkage by the receiving Link Endpoint this field indicates the lifetime policy in use for the Node which has been dynamically created at the time of the attach. In this case the address field will contain the address of the created Node. The generated address SHOULD include the Link name and Session-name or client-id in some recognizable form for ease of traceability.
The lifetime of the generated node is controlled by the policy specified in the dynamic field in the Target of the local Linkage of the receiving Link Endpoint. Definitionally, the lifetime will never be less than the lifetime of the link which caused its creation.

`capabilities` *the extension capabilities the sender supports/desires*

4.5.5 Terminus Expiry Policy

Expiry policy for a Terminus.

```
<type name="terminus-expiry-policy" class="restricted" source="symbol">
    <choice name="session" value="session"/>
    <choice name="connection" value="connection"/>
    <choice name="never" value="never"/>
</type>
```

Determines when the expiry timer of a Terminus starts counting down from the timeout value.

Valid Values

session

The expiry timer starts when the last associated session is ended.

connection

The expiry timer starts when last associated connection is closed.

never

The Terminus never expires.

4.5.6 Std Dist Mode

Link distribution policy.

```
<type name="std-dist-mode" class="restricted" source="symbol" provides="distribution-mode">
    <choice name="move" value="move"/>
    <choice name="copy" value="copy"/>
</type>
```

Policies for distributing Messages when multiple Links are connected to the same Node.

Valid Values

move

once successfully transferred over the Link, the Message will no longer be available to other Links from the same Node

copy

once successfully transferred over the Link, the Message is still available for other Links from the same Node

4.5.7 Filter

The predicate to filter the Messages admitted onto the Link.

```
<type name="filter" class="composite" source="list">
    <descriptor name="amqp:filter:list" code="0x00000000:0x0000002a"/>
    <field name="type" type="symbol" mandatory="true"/>
    <field name="predicate" type="*" requires="predicate"/>
</type>
```

Field Details

`type`	*the type of the filter*
`predicate`	*the filter predicate*

4.5.8 Filter Set

```
<type name="filter-set" class="restricted" source="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

4.5.9 Delete On Close

Lifetime of dynamic Node scoped to lifetime of link which caused creation.

```
<type name="delete-on-close" class="composite" source="list" provides="lifetime-policy">
    <descriptor name="amqp:delete-on-close:list" code="0x00000000:0x0000002b"/>
    <field name="options" type="options"/>
</type>
```

A Node dynamically created with this lifetime policy will be deleted at the point that the Link which caused its creation ceases to exist.

Field Details

`options`	*options map*

4.5.10 Delete On No Links

Lifetime of dynamic Node scoped to existence of links to the Node.

```
<type name="delete-on-no-links" class="composite" source="list" provides="lifetime-policy">
    <descriptor name="amqp:delete-on-no-links:list" code="0x00000000:0x0000002c"/>
    <field name="options" type="options"/>
</type>
```

A Node dynamically created with this lifetime policy will be deleted at the point that there remain no Links for which the node is either the source or target.

Field Details

`options`	*options map*

4.5.11 Delete On No Messages

Lifetime of dynamic Node scoped to existence of messages on the Node.

```
<type name="delete-on-no-messages" class="composite" source="list" provides="lifetime-policy">
    <descriptor name="amqp:delete-on-no-messages:list" code="0x00000000:0x0000002d"/>
    <field name="options" type="options"/>
</type>
```

A Node dynamically created with this lifetime policy will be deleted at the point that the Link which caused its creation no longer exists and there remain no Messages at the Node.

Field Details

`options`	*options map*

4.5.12 Delete On No Links Or Messages

Lifetime of Node scoped to existence of messages on or links to the Node.

```
<type name="delete-on-no-links-or-messages" class="composite" source="list" provides="lifetime-policy">
    <descriptor name="amqp:delete-on-no-links-or-messages:list" code="0x00000000:0x0000002e"/>
    <field name="options" type="options"/>
</type>
```

A Node dynamically created with this lifetime policy will be deleted at the point that the there are no Links which have this Node as their source or target, and there remain no Messages at the Node.

Field Details

`options`	*options map*

4.5.13 MESSAGE-FORMAT

MESSAGE-FORMAT: the format + revision for the messages defined by this document	0

This value goes into the message-format field of the transfer frame when transferring messages of the format defined herein.

Book 5

Transactions

5.1 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 coordinator. Should the coordinator fail to perform the `declare` or `discharge` as specified in the message, the message will be rejected with a `transaction-errors` indicating the failure condition that occurred.

The `declare` and `discharge` message bodies MUST be encoded as a single section with a section-codes of *amqp-data* containing exactly one encoded AMQP value of the appropriate described type.

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.

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.

5.1.1 Transactional Acquisition

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.

While the *txn-id* is cleared when the transaction is discharged, this does not affect the level of outstanding credit. To prevent the sending link endpoint from acquiring outside of any transaction, the *controller* SHOULD ensure there is no outstanding credit at the sender before it discharges the transaction. The *controller* may do this by either setting the drain mode of the sending link endpoint to *true* before discharging the transaction, or by reducing the *link-credit* to zero, and waiting to hear back that the sender has seen this state change.

If a transaction is discharged at a point where a message has been transactionally acquired, but has not been fully sent (i.e. the delivery of the message will require more than one transfer frame and at least one, but not all, such frames have been sent) then the resource MUST interpret this to mean that the fate of the acquisition is fully decided by the discharge. If the `discharge` unambiguously determines the *outcome* of the transfer (e.g. the `discharge` indicates the failure of the transaction, or the transfer has a single pre-determined outcome), the resource MAY decide not to abort sending the remainder of the message.

5.2 Coordination

5.2.1 Coordinator

Target for communicating with a transaction coordinator.

```
<type name="coordinator" class="composite" source="list" provides="target">
    <descriptor name="amqp:coordinator:list" code="0x00000000:0x00000030"/>
    <field name="options" type="options"/>
    <field name="capabilities" type="symbol" requires="txn-capabilities" multiple="true"/>
</type>
```

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

`options`	*options map*
`capabilities`	

5.2.2 Declare

Message body for declaring a transaction id.

```
<type name="declare" class="composite" source="list">
    <descriptor name="amqp:declare:list" code="0x00000000:0x00000031"/>
    <field name="options" type="options"/>
    <field name="global-id" type="*" requires="global-tx-id"/>
</type>
```

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 map*

`global-id` *global transaction id*

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.

5.2.3 Discharge

Message body for discharging a transaction.

```
<type name="discharge" class="composite" source="list">
    <descriptor name="amqp:discharge:list" code="0x00000000:0x00000032"/>
    <field name="options" type="options"/>
    <field name="fail" type="boolean"/>
</type>
```

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 map*
`fail`	*indicates the transaction should be rolled back*

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.

5.2.4 Txn Capabilities

Symbols indicating (desired/available) capabilities of a transaction coordinator.

```
<type name="txn-capabilities" class="restricted" source="symbol" provides="txn-capabilities">
    <choice name="local-txn" value="amqp:local-transactions"/>
    <choice name="distributed-txn" value="amqp:distributed-transactions"/>
    <choice name="promotable-txn" value="amqp:promotable-transactions"/>
    <choice name="multi-txns-per-ssn" value="amqp:multi-txns-per-ssn"/>
    <choice name="multi-ssns-per-txn" value="amqp:multi-ssns-per-txn"/>
    <choice name="multi-conns-per-txn" value="amqp:multi-conns-per-txn"/>
</type>
```

Valid Values

amqp:local-transactions

Support local transactions.

amqp:distributed-transactions

Support AMQP Distributed Transactions.

amqp:promotable-transactions

Support AMQP Promotable Transactions.

amqp:multi-txns-per-ssn

Support multiple active transactions on a single session.

amqp:multi-ssns-per-txn

Support transactions whose txn-id is used across sessions on one connection.

amqp:multi-conns-per-txn

Support transactions whose txn-id is used across different connections.

5.2.5 Transaction Errors

Symbols used to indicate transaction errors.

```
<type name="transaction-errors" class="restricted" source="symbol" provides="error">
    <choice name="unknown-id" value="amqp:transaction:unknown-id"/>
    <choice name="transaction-rollback" value="amqp:transaction:rollback"/>
    <choice name="transaction-timeout" value="amqp:transaction:timeout"/>
</type>
```

Valid Values

amqp:transaction:unknown-id
: The specified txn-id does not exist.

amqp:transaction:rollback
: The transaction was rolled back for an unspecified reason.

amqp:transaction:timeout
: The work represented by this transaction took too long.

Book 6

Security

6.1 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.

6.2 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 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 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%d2.1.0.0  --------->
               <---------  AMQP%d2.1.0.0
                    :
                    :
            <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. If it does so, then it is implementation-specific what happens if this differs to hostname in the `sasl-init` and `open` frame frames.

  This field can be used by AMQP proxies to determine the correct back-end service to connect the client to, and to determine the domain to validate the client's credentials against if TLS client certificates are being used.
- The TLS client MUST validate the certificate presented by the TLS server.

6.2.1 Alternative Establishment

In certain situations, such as connecting through firewalls, it may not be possible to establish a TLS security layer using tunnelling. This might be because a deep packet inspecting firewall sees the first few bytes of the connection 'as not being TLS'.

As an alternative, implementations MAY run a pure TLS server, i.e., one that does not expect the tunnel negotiation handshake. The IANA service name for this is amqps and the port is SECURE-PORT (5671). Implementations may also choose to run this pure TLS server on other ports, should this be operationally required (e.g. to tunnel through a legacy firewall that only expects TLS traffic on port 443).

6.3 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 three, 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"  |   %d3   |  major  |  minor  | revision |
+----------+---------+---------+---------+----------+
```

Other than using a protocol id of three, 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%d3.1.0.0  --------->
               <---------  AMQP%d3.1.0.0
                    :
                    :
            <SASL negotiation>
                    :
                    :
AMQP%d0.1.0.0  --------->                 (over SASL secured connection)
               <---------  AMQP%d0.1.0.0
         open  --------->
               <---------  open
```

6.3.1 SASL Frames

SASL is negotiated using framing. A SASL frame has a type code of 0x01. Bytes 6 and 7 of the header are ignored. Implementations SHOULD set these to 0x00. The extended header is ignored. Implementations SHOULD therefore set DOFF to 0x02.

```
                 type: 0x01 - SASL frame

                +0       +1       +2       +3
             +-----------------------------------+ -.
           0 |                SIZE               |  |
             +-----------------------------------+  |---> Frame Header
           4 |  DOFF  |  TYPE  |   <IGNORED>*1   |  |      (8 bytes)
             +-----------------------------------+ -'
             +-----------------------------------+ -.
           8 |                ...                |  |
             .                                   .  |---> Extended Header
             .             <IGNORED>*2           .  |  (DOFF * 4 - 8) bytes
             |                ...                |  |
             +-----------------------------------+ -'
             +-----------------------------------+ -.
      4*DOFF |                                   |  |
             .                                   .  |
             .                                   .  |
             .   Sasl Mechanisms / Sasl Init     .  |
             .  Sasl Challenge / Sasl Response   .  |---> Frame Body
             .          Sasl Outcome             .  |  (SIZE - DOFF * 4) bytes
             .                                   .  |
             .                                   .  |
             .                          _________|  |
             |                ...       |           |
             +--------------------------+          -'

              *1 SHOULD be set to 0x0000
              *2 Ignored, so DOFF should be set to 0x02
```

Figure 6.1: SASL Frame

The maximum size of a SASL frame is defined by MIN-MAX-FRAME-SIZE. There is no mechanism within the SASL negotiation to negotiate a different size. The frame body of a SASL frame may contain exactly one AMQP type, whose type encoding must have provides="frame" . Receipt of an empty frame is an irrecoverable error.

6.3.2 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 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.
```

Figure 6.2: SASL Exchange

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.

6.3.3 Security Frame Bodies

6.3.3.1 Sasl Mechanisms

Advertise available sasl mechanisms.

```
<type name="sasl-mechanisms" class="composite" source="list" provides="sasl-frame">
    <descriptor name="amqp:sasl-mechanisms:list" code="0x00000000:0x00000040"/>
    <field name="options" type="options"/>
    <field name="sasl-server-mechanisms" type="string" multiple="true" mandatory="true"/>
</type>
```

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

Field Details

`options` *options map*

`sasl-server-mechanisms` *supported sasl mechanisms*

A list of the sasl security mechanisms supported by the sending peer. It is invalid for this list to be null or empty. If the sending peer does not require its partner to authenticate with it, then it should send a list of one element with its value as the SASL mechanism *ANONYMOUS*. The server mechanisms are ordered in decreasing level of preference.

6.3.3.2 Sasl Init

Initiate sasl exchange.

```
<type name="sasl-init" class="composite" source="list" provides="sasl-frame">
    <descriptor name="amqp:sasl-init:list" code="0x00000000:0x00000041"/>
    <field name="options" type="options"/>
    <field name="mechanism" type="string" mandatory="true"/>
    <field name="initial-response" type="binary"/>
    <field name="hostname" type="string"/>
</type>
```

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

Field Details

`options`	*options map*

`mechanism` *selected security mechanism*

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` *security response data*

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

`hostname` *the name of the target host*

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.
This field can be used by AMQP proxies to determine the correct back-end service to connect the client to, and to determine the domain to validate the client's credentials against.
This field may already have been specified by the server name indication extension as described in RFC-4366, if a TLS layer is used, in which case this field SHOULD be null or contain the same value. It is undefined what a different value to those already specific means.

6.3.3.3 Sasl Challenge

Security mechanism challenge.

```
<type name="sasl-challenge" class="composite" source="list" provides="sasl-frame">
    <descriptor name="amqp:sasl-challenge:list" code="0x00000000:0x00000042"/>
    <field name="options" type="options"/>
    <field name="challenge" type="binary" mandatory="true"/>
</type>
```

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

Field Details

`options`	*options map*
`challenge`	*security challenge data*

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

6.3.3.4 Sasl Response

Security mechanism response.

```
<type name="sasl-response" class="composite" source="list" provides="sasl-frame">
    <descriptor name="amqp:sasl-response:list" code="0x00000000:0x00000043"/>
    <field name="options" type="options"/>
    <field name="response" type="binary" mandatory="true"/>
</type>
```

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

Field Details

`options` *options map*

`response` *security response data*

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

6.3.3.5 Sasl Outcome

Indicates the outcome of the sasl dialog.

```
<type name="sasl-outcome" class="composite" source="list" provides="sasl-frame">
    <descriptor name="amqp:sasl-outcome:list" code="0x00000000:0x00000044"/>
    <field name="options" type="options"/>
    <field name="code" type="sasl-code" mandatory="true"/>
    <field name="additional-data" type="binary"/>
</type>
```

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` *options map*

`code` *indicates the outcome of the sasl dialog*

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

`additional-data` *additional data as specified in RFC-4422*

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.

6.3.3.6 Sasl Code

Codes to indicate the outcome of the sasl dialog.

```
<type name="sasl-code" class="restricted" source="ubyte">
    <choice name="ok" value="0"/>
    <choice name="auth" value="1"/>
    <choice name="sys" value="2"/>
    <choice name="sys-perm" value="3"/>
    <choice name="sys-temp" value="4"/>
</type>
```

Valid Values

0

Connection authentication succeeded.

1

Connection authentication failed due to an unspecified problem with the supplied credentials.

2

Connection authentication failed due to a system error.

3

Connection authentication failed due to a system error that is unlikely to be corrected without intervention.

4

Connection authentication failed due to a transient system error.