AMQP v1.0 (revision 0)

23 Aug 2011

Confidential DRAFT

URL: http://svn.amqp.org/svn/amqp/trunk
Revision: 1346

Copyright Notice

Trademarks

Link to full AMQP specification

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

Contents

Introduction 6

1 Types 9
1.1 Type System 9
1.1.1 Primitive Types 9
1.1.2 Described Types 10
1.1.3 Descriptor Values 10
1.2 Type Encodings 10
1.2.1 Fixed Width 13
1.2.2 Variable Width 17
1.2.3 Compound 18
1.2.4 Array 20
1.2.5 List Of Encodings 20
1.3 Composite Types 21
1.3.1 List Encoding 22

2 Transport 24
2.1 Transport 24
2.2 Version Negotiation 26
2.3 Framing 28
2.3.1 Frame Layout 28
2.3.2 AMQP Frames 29
2.4 Connections 30
2.4.1 Opening A Connection 31
2.4.2 Pipelined Open 31
2.4.3 Closing A Connection 32
2.4.4 Simultaneous Close 32
2.4.5 Idle Time Out Of A Connection 33
2.4.6 Connection States 33
2.4.7 Connection State Diagram 34
2.5 Sessions 35
2.5.1 Establishing A Session 36
2.5.2 Ending A Session 36
2.5.3 Simultaneous End 37
2.5.4 Session Errors 37
2.5.5 Session States 38
2.5.6 Session Flow Control 39
2.6 Links 40
2.6.1 Naming A Link 41
2.6.2 Link Handles 41
2.6.3 Establishing Or Resuming A Link 42

2.6.4 Detaching And Reattaching A Link . . . 44
2.6.5 Link Errors . . . 45
2.6.6 Closing A Link . . . 46
2.6.7 Flow Control . . . 46
2.6.8 Synchronous Get . . . 48
2.6.9 Asynchronous Notification . . . 49
2.6.10 Stopping A Link . . . 49
2.6.11 Messages . . . 50
2.6.12 Transferring A Message . . . 50
2.6.13 Resuming Deliveries . . . 54
2.6.14 Transferring Large Messages . . . 54
2.7 Performatives . . . 55
2.7.1 Open . . . 55
2.7.2 Begin . . . 57
2.7.3 Attach . . . 59
2.7.4 Flow . . . 61
2.7.5 Transfer . . . 62
2.7.6 Disposition . . . 65
2.7.7 Detach . . . 66
2.7.8 End . . . 66
2.7.9 Close . . . 67
2.8 Definitions . . . 67
2.8.1 Role . . . 67
2.8.2 Sender Settle Mode . . . 67
2.8.3 Receiver Settle Mode . . . 68
2.8.4 Handle . . . 68
2.8.5 Seconds . . . 68
2.8.6 Milliseconds . . . 69
2.8.7 Delivery Tag . . . 69
2.8.8 Delivery Number . . . 69
2.8.9 Transfer Number . . . 69
2.8.10 Sequence No . . . 69
2.8.11 Message Format . . . 69
2.8.12 IETF Language Tag . . . 69
2.8.13 Fields . . . 70
2.8.14 Error . . . 70
2.8.15 AMQP Error . . . 70
2.8.16 Connection Error . . . 72
2.8.17 Session Error . . . 72
2.8.18 Link Error . . . 73
2.8.19 Constant Definitions . . . 74

3 Messaging **75**
3.1 Introduction . . . 75
3.2 Message Format . . . 75
3.2.1 Header . . . 76
3.2.2 Delivery Annotations . . . 78
3.2.3 Message Annotations . . . 78
3.2.4 Properties . . . 78
3.2.5 Application Properties . . . 80
3.2.6 Data . . . 81
3.2.7 AMQP Sequence . . . 81
3.2.8 AMQP Value . . . 81

3.2.9 Footer . . . 81
3.2.10 Annotations . . . 81
3.2.11 Message ID ULong . . . 82
3.2.12 Message ID UUID . . . 82
3.2.13 Message ID Binary . . . 82
3.2.14 Message ID String . . . 82
3.2.15 Address String . . . 82
3.2.16 MESSAGE-FORMAT . . . 82
3.3 Distribution Nodes . . . 82
3.3.1 Message States . . . 82
3.4 Delivery State . . . 83
3.4.1 Received . . . 84
3.4.2 Accepted . . . 84
3.4.3 Rejected . . . 85
3.4.4 Released . . . 85
3.4.5 Modified . . . 86
3.4.6 Resuming Deliveries Using Delivery States . . . 86
3.5 Sources and Targets . . . 90
3.5.1 Filtering Messages . . . 90
3.5.2 Distribution Modes . . . 90
3.5.3 Source . . . 91
3.5.4 Target . . . 93
3.5.5 Terminus Durability . . . 94
3.5.6 Terminus Expiry Policy . . . 94
3.5.7 Standard Distribution Mode . . . 95
3.5.8 Filter Set . . . 95
3.5.9 Node Properties . . . 95
3.5.10 Delete On Close . . . 96
3.5.11 Delete On No Links . . . 96
3.5.12 Delete On No Messages . . . 96
3.5.13 Delete On No Links Or Messages . . . 96

4 Transactions 98
4.1 Transactional Messaging . . . 98
4.2 Declaring a Transaction . . . 98
4.3 Discharging a Transaction . . . 99
4.4 Transactional Work . . . 100
4.4.1 Transactional Posting . . . 101
4.4.2 Transactional Retirement . . . 101
4.4.3 Transactional Acquisition . . . 102
4.4.4 Interaction Of Settlement With Transactions . . . 103
4.4.4.1 Transactional Posting . . . 103
4.4.4.2 Transactional Retirement . . . 104
4.4.4.3 Transactional Acquisition . . . 104
4.5 Coordination . . . 105
4.5.1 Coordinator . . . 105
4.5.2 Declare . . . 105
4.5.3 Discharge . . . 106
4.5.4 Transaction ID . . . 106
4.5.5 Declared . . . 106
4.5.6 Transactional State . . . 106
4.5.7 Transaction Capability . . . 107
4.5.8 Transaction Error . . . 107

5 Security 109
5.1 Security Layers 109
5.2 TLS 109
5.2.1 Alternative Establishment 110
5.2.2 Constant Definitions 110
5.3 SASL 111
5.3.1 SASL Frames 111
5.3.2 SASL Negotiation 112
5.3.3 Security Frame Bodies 113
5.3.3.1 SASL Mechanisms 113
5.3.3.2 SASL Init 113
5.3.3.3 SASL Challenge 114
5.3.3.4 SASL Response 114
5.3.3.5 SASL Outcome 114
5.3.3.6 SASL Code 115
5.3.4 Constant Definitions 115

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.

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

Book 1

Types

1.1 Type System

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

1.1.1 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 inclusive
ushort	integer in the range 0 to 2^{16} - 1 inclusive
uint	integer in the range 0 to 2^{32} - 1 inclusive
ulong	integer in the range 0 to 2^{64} - 1 inclusive
byte	integer in the range $-(2^7)$ to 2^7 - 1 inclusive
short	integer in the range $-(2^{15})$ to 2^{15} - 1 inclusive
int	integer in the range $-(2^{31})$ to 2^{31} - 1 inclusive
long	integer in the range $-(2^{63})$ to 2^{63} - 1 inclusive
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
array	a sequence of values of a single type

1.1.2 Described Types

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

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

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

1.1.3 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. For numeric ids the assigned domain-id will be equal to the IANA Private Enterprise Number (PEN) of the requesting organisation (`http://www.iana.org/assignments/enterprise-numbers`) with domain-id 0 reserved for descriptors defined in the AMQP Specification.

Descriptors are then assigned within each domain according to the following rules:

symbolic descriptors

$<$*domain*$>$**:**$<$*name*$>$

numeric descriptors

(*domain-id* << 32) | *descriptor-id*

1.2 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 1.1: 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 1.2: 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 constructor

  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 1.3: 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.

1.2.1 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
	0x56	fixed-width, 1 byte value	*boolean with the octet 0x00 being false and octet 0x01 being true*
`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 to 255 inclusive*
`uint0`	0x43	fixed-width, 0 byte value	*the uint value 0*

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 to 255 inclusive*
`ulong0`	0x44	fixed-width, 0 byte value	*the ulong value 0*

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	*signed integer value in the range -128 to 127 inclusive*

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	*signed long value in the range -128 to 127 inclusive*

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*

Represents an approximate point in time using the Unix time_t encoding of UTC, but with a precision of milliseconds. For example, 1311704463521 represents the moment 2011-07-26T18:21:03.521Z.

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*

1.2.2 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)*
`str32-utf8`	0xb1	variable-width, 4 byte size	*up to* 2^{32} *- 1 octets worth of UTF-8 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*

1.2.3 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**
`list0`	0x45	fixed-width, 0 byte value	*the empty list (i.e. the list with no elements)*
`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*

Type: `map`

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

A map is encoded as a compound value where the constituent elements form alternating key value pairs.

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

Map encodings must contain an even number of items (i.e. an equal number of keys and values). A map in which there exist two identical key values is invalid. Unless known to be otherwise, maps must be considered to be ordered - that is the order of the key-value pairs is semantically important and two maps which are different only in the order in which their key-value pairs are encoded are not equal.

Encoding	**Code**	**Category**	**Description**
`map8`	0xc1	variable-width, 1 byte size	*up to* 2^8 *- 1 octets of encoded map data*
`map32`	0xd1	variable-width, 4 byte size	*up to* 2^{32} *- 1 octets of encoded map data*

1.2.4 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
```

Type: `array`

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

Encoding	Code	Category	Description
`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*

1.2.5 List Of Encodings

Type	Encoding	Code	Category	Description
null		0x40	fixed/0	the null value
boolean		0x56	fixed/1	boolean with the octet 0x00 being false and octet 0x01 being true
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 to 255 inclusive
uint	uint0	0x43	fixed/0	the uint value 0
ulong		0x80	fixed/8	64-bit unsigned integer in network byte order
ulong	smallulong	0x53	fixed/1	unsigned long value in the range 0 to 255 inclusive
ulong	ulong0	0x44	fixed/0	the ulong value 0
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

Type	Encoding	Code	Category	Description
int	smallint	0x54	fixed/1	signed integer value in the range -128 to 127 inclusive
long		0x81	fixed/8	64-bit two's-complement integer in network byte order
long	smalllong	0x55	fixed/1	signed long value in the range -128 to 127 inclusive
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	str32-utf8	0xb1	variable/4	up to 2^{32} - 1 octets worth of UTF-8 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	list0	0x45	fixed/0	the empty list (i.e. the list with no elements)
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
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
array	array8	0xe0	array/1	up to 2^8 - 1 array elements with total size less than 2^8 octets
array	array32	0xf0	array/4	up to 2^{32} - 1 array elements with total size less than 2^{32} octets

1.3 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 1.4: 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. Multiple values are represented by the use of an array where the type of the elements in the array is the type defined in the field definition. Note that a null value and a zero-length array (with a correct type for its elements) both describe an absence of a value and should be treated as semantically identical.

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 an array with no entries are invalid).

1.3.1 List Encoding

AMQP composite values are encoded 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 0x40 0x03  title  authors  isbn
      |                 |                            |       |       |
      |      identifies composite type               |       |       |
      |                                              |       |     0x40
    sym8                      +----------------------+       |       |
  (symbol)                    |                              |  null value
               +--------------+----------------+             |
               |                               |             |
               0xA1 0x15 "AMQP for & by Dummies"             |
                                                             |
+------------------------------------------------------------+-----+
|                                                                  |
0xE0 0x25 0x02 0xA1 0x0E "Rob J. Godfrey" 0x13 "Rafael H. Schloming"
      |    |    |   |                   | |                        |
    size   |    |   +---------+---------+ +-----------+------------+
           |    |             |                       |
         count  |       first element          second element
                |
        element constructor
```

Figure 1.5: Example Composite Value

Book 2

Transport

2.1 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 section 2.6 Links) 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 nth 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 (see section 2.3 Framing).

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 2.1: 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 section 2.6 Links) 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 section 2.7 Performatives. The following table lists all frame bodies and defines which endpoints handle them.

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

```
Key:
     H: handled by the endpoint

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

2.2 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 2.2: Version Negotiation Examples

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

The protocol id is not a part of the protocol version and thus the rule above regarding the highest supported version does not apply. A client might request use of a protocol id that is unacceptable to a server - for example, it might request a raw AMQP connection when the server is configured to require a TLS or SASL security layer (See section 5.1 Security Layers). In this case, the server MUST send a protocol header with an **acceptable** protocol id (and version) and then close the socket. It MAY choose any protocol id.

```
TCP Client                               TCP Server
======================================================
AMQP%d0.1.0.0       ------------->
                    <-------------       AMQP%d3.1.0.0
                                         *TCP CLOSE*
------------------------------------------------------
      Server rejects Connection for: AMQP, protocol=0,
      major=1, minor=0, revision=0, Server responds
      that it requires: SASL security layer, protocol=3,
      major=1, minor=0, revision=0
```

Figure 2.3: Protocol ID Rejection Example

2.3 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 preceding 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.

2.3.1 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 a SASL frame, see section 5.3 SASL).

2.3.2 AMQP Frames

Bytes 6 and 7 of an AMQP Frame contain the Channel number (see section 2.1 Transport). The frame body is defined as a *performative* followed by an opaque *payload*. The performative MUST be one of those defined in section 2.7 Performatives and is encoded as a described type in the AMQP type system. The remaining bytes in the frame body form the payload for that frame. The presence and format of the payload is defined by the semantics of the given performative.

```
                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 |           PERFORMATIVE:           |  |
         .      Open / Begin / Attach        .  |
         .   Flow / Transfer / Disposition   .  |
         .      Detach / End / Close         .  |
         |-----------------------------------|  |
         .                                   .  |---> Frame Body
         .                                   .  |  (SIZE - DOFF * 4) bytes
         .             PAYLOAD               .  |
         .                                   .  |
         .                           ________|  |
         |                ...       |           |
         +--------------------------+          -'
```

An AMQP frame with no body may be used to generate artificial traffic as needed to satisfy any negotiated idle time-out interval. See 2.4.5 Idle Time-out of a Connection.

2.4 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 (see section 2.5 Sessions).

2.4.1 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 512 (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

                ...
```

2.4.2 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.

2.4.3 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 (in order to guard against erroneously or maliciously implemented partners, a peer SHOULD implement a timeout to give its partner a reasonable time to receive and process the close before giving up and simply closing the underlying transport mechanism). 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 pipelined in a single batch with the corresponding `open`.

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

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

Implementations SHOULD NOT expect to be able to reuse open TCP sockets after `close` performatives have been exchanged. There is no requirement for an implementation to read from a socket after a `close` performative has been received.

2.4.4 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
```

2.4.5 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.

2.4.6 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.
DISCARDING	The DISCARDING state is a variant of the CLOSE_SENT state where the `close` is triggered by an error. In this case any incoming frames on the connection MUST be silently discarded until the peer's `close` frame is received.
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.

2.4.7 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>
* Also could be DISCARDING if an error condition
  triggered the CLOSE
```

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
DISCARDING	-	*	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
```

2.5 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 delivery sequence scoped to the session. This common sequence allows endpoints to efficiently refer to sets of deliveries regardless of the originating link. This is of particular benefit when a single application is receiving messages 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.

2.5.1 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.

To make it easier to monitor AMQP sessions, it is recommended that implementations always assign the lowest available unused channel number.

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)
                                  /
                             <---+

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

2.5.2 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.
```

2.5.3 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.
```

2.5.4 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              ---------->
                     ...
================================================
```

2.5.5 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 2.4: 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.

2.5.6 Session Flow Control

The Session Endpoint assigns each outgoing `transfer` frame an implicit *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 implicit 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 transfer-id 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 increment the next-incoming-id to match the implicit 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:

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

2.6 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* deliveries. 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 *Delivery 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 Delivery State. The Link Endpoint assigns each Message a unique *delivery-id* from a Session scoped sequence. These delivery-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 re-establishing communication is referred to as *resuming* a Link.

The original Link Endpoint state is not necessary for resumption of a Link. Only the unsettled Delivery 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 Delivery 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.

2.6.1 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.

A link's 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 (ordered) tuple *(A,B,<name>)*.

Consequently, a link may only be active in one connection at a time. If an attempt is made to attach the link subsequently when it is not suspended, then the link can be 'stolen', i.e. the second attach succeeds and the first attach must then be closed with a link error of `stolen`. This behavior ensures that in the event of a connection failure occurring and being noticed by one party, that re-establishment has the desired effect.

2.6.2 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        |
+-------------------+                             +-------------------+
```

2.6.3 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 2.5: 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 2.6: Refusing a Link

If either end of the Link is already associated with a Terminus, the `attach` frame MUST include its unsettled delivery 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 2.7: 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`.

Note that a peer MUST take responsibility for verifying that the remote terminus meets its requirements. The remote peer SHOULD NOT attempt to pre-empt whether the terminus will meet the requirements of its partner. This is equally true both for creating and resuming links.

```
          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 2.8: 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 2.9: Recovering a Link

2.6.4 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 deliveries on the link referenced via delivery-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.
```

2.6.5 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 delivery-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. In order to disambiguate the resume request from a pipelined re-attach the resuming `attach` performative MUST contain a non-null value for its unsettled field. Receipt of a pipelined `attach` MUST result in the session being terminated with an `errant-link` session-error.

2.6.6 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 2.10: 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.

2.6.7 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 messages to consume the currently available *link-credit*, or insufficient *link-credit* to send available messages:

delivery-count The *delivery-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 *delivery-count* may be increased. This identifies a *delivery-limit* that may be computed by adding the *link-credit* to the *delivery-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 *delivery-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:

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

In the event that the receiver does not yet know the *delivery-count*, i.e. $\textit{delivery-count}_{\mathrm{rcv}}$ is unspecified, the Sender MUST assume that the $\textit{delivery-count}_{\mathrm{rcv}}$

is the first *delivery-count*$_{\text{snd}}$ sent from Sender to Receiver, i.e. the *delivery-count*$_{\text{snd}}$ specified in the flow state carried by the initial `attach` frame from the Sender to the Receiver.

Additionally, whenever the Sender increases *delivery-count*, it MUST decrease *link-credit* by the same amount in order to maintain the *delivery-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 messages are available to consume the current link-credit. If set, the Sender will (after sending all available messages) advance the delivery-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 delivery-count is the same as or greater than the delivery-limit, it is illegal to send more messages. If the link-credit is reduced by the Receiver when transfers are in-flight, the Receiver MAY either handle the excess messages 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 messages, the Sender MUST advance its delivery-count until link-credit is zero, and send its updated `flow` state to the Receiver.

The delivery-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 *delivery-count*. This value is communicated in the initial `attach` frame. The Receiver initializes its *delivery-count* upon receiving the Sender's `attach`.

```
                                flow state
                                     |
                                     | modifies
                                     |
+------------------+                                +------------------+
|      Sender      |    .-----------------------.   |     Receiver     |
+------------------+     attach, transfer, flow     +------------------+
| delivery-count   |------------------------------->| delivery-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, ...)      ---->
```

2.6.8 Synchronous Get

A synchronous get of a message 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 messages 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 message 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 message is available within the timeout, it will
            arrive at this point.
        (2) If a message is not available within the timeout, the
            drain flag will ensure that the Sender promptly advances the
            delivery-count until link-credit is consumed.
```

2.6.9 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 delivery-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 message rate for the Link is limited by the
  rate at which the Receiver updates the delivery-limit by
  issuing link-credit.
```

2.6.10 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
        =================================================================
                                                  ...
                                              <---------- transfer(...)
        flow(...,                             ---+   +--- transfer(...)
             link-credit=0,                       \ /
             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.
```

2.6.11 Messages

The transport layer assumes as little as possible about Messages and allows alternative Message representations to be layered above. Message data is carried as the payload in frames containing the `transfer` performative. Messages can be fragmented across several `transfer` frames as indicated by the more flag of the `transfer` performative.

2.6.12 Transferring A Message

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

Upon initiating a transfer, the application 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 delivery-id assigned by the session. The delivery-id is then used to refer to the delivery-tag in all subsequent interactions on that session.

For simplicity the delivery-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, ...)
|   DT -> (local: S_0, |----------------------------------------------->
|          remote: ?)  |
|   ...                |
+----------------------+
```

Figure 2.11: 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, ...)                   |   ...                |
  ---------------------------------------------->|   DT -> (local: S_1, |
                                                 |          remote: S_0)|
                                                 |   ...                |
                                                 +----------------------+
```

Figure 2.12: Initial Receipt

Once notified of the received message data, the application processes the message, indicating the updated delivery state to the link endpoint as desired. Applications may wish to classify delivery 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 delivery 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 2.13: Indication of Non-Terminal State

Once the receiving application has finished processing the message, it indicates to the link endpoint a *terminal* delivery state that reflects the outcome of the application processing (successful or otherwise) and thus the outcome which the Receiver wishes to occur at the Sender. This state is 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 2.14: Indication of Presumptive Terminal State

Upon receiving the updated delivery state from the Receiver, the Sender will, if it has not already spontaneously attained a terminal state, update its view the 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 2.15: Receipt of Terminal State

The sending application will then typically perform some action based on this terminal state and then settle the delivery, causing the Sender to remove the delivery-tag from its unsettled map. The Sender will then send its final delivery state along with an indication that the delivery 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 delivery upon link recovery. When the Sender re-attaches the Receiver will examine the unsettled state of the Sender (i.e. what has **not** been forgotten) and from this can derive that the delivery in question has been settled (since its tag will not be in the unsettled state).

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

Figure 2.16: Indication of Settlement

When the Receiver finds out that the Sender has settled the delivery, 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 2.17: 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 delivery is settled. The Receiver will then remove the delivery-tag from its unsettled map. Note that because the Receiver knows that the delivery is already settled at the Sender, it makes no effort to notify the other endpoint that it is settling the delivery.

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

Figure 2.18: Final Settlement

As alluded to above, it is possible for the sending application to transition a delivery to a terminal state at the Sender spontaneously (i.e. not as a consequence of a disposition that has been received from the Receiver). In this case the Sender should send a disposition to the Receiver, but not settle until the Receiver confirms, via a disposition in the opposite direction, that it has updated the state at its endpoint.

This set of exchanges illustrates the basic principals of message transfer. While a delivery is unsettled the endpoints exchange the current state of the delivery. 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 delivery, 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 delivery *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 delivery and will therefore make no further attempts to send it. Should this delivery make it to the Receiver, the Receiver clearly has no obligation to respond with updates of the Receiver's delivery state, as they would be meaningless and ignored by the Sender.

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

Figure 2.19: 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 delivery was 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 2.20: 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 delivery. An application may also choose to settle at an endpoint independently of its delivery state, for example the sending application may choose to settle a delivery due to the message ttl expiring regardless of whether the Receiver has reached a terminal state.

2.6.13 Resuming Deliveries

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

Deliveries that only the Source considers unsettled
Deliveries in this category MAY be resumed at the discretion of the sending application. If the sending application marks the resend attempt as a resumed delivery then it MUST be ignored by the receiver. (This allows the sender to pipeline resumes without risk of duplication at the sender).

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

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

Note that in the case where an endpoint indicates that the unsettled map is incomplete, the absence of an entry in the unsettled map is not an indication of settlement. In this case the two endpoints must reduce the levels of unsettled state as much as they can by the Sender resuming and/or settling transfers that it observes that the Receiver considers unsettled. Upon completion of this reduction of state, the two parties must suspend and re-attempt to resume the link. Only when both sides have complete unsettled maps may new unsettled state be created by the sending of non-resuming transfers.

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

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

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

2.6.14 Transferring Large Messages

Each `transfer` frame may carry an arbitrary amount of message data up to the limit imposed by the maximum frame size. For Messages that are too large to fit within the maximum frame size, additional data 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 deliver a Message by setting the abort flag on the last `transfer`. In this case the receiver MUST discard the Message data that was transferred prior to the abort.

```
              +------------+  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 2.21: 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 2.22: Incoming Fragmentation State Diagram

2.7 Performatives

2.7.1 Open

Negotiate Connection parameters.

```
<type name="open" class="composite" source="list" provides="frame">
    <descriptor name="amqp:open:list" code="0x00000000:0x00000010"/>
    <field name="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="ietf-language-tag" multiple="true"/>
    <field name="incoming-locales" type="ietf-language-tag" 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

`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 512 (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 (see subsection 2.8.12 IETF Language Tag). 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.
A list of commonly defined connection capabilities and their meanings can be found here: `http://www.amqp.org/specification/1.0/connection-capabilities`.

`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.
A list of commonly defined connection properties and their meanings can be found here: `http://www.amqp.org/specification/1.0/connection-properties`

2.7.2 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="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 has begun on the channel.

Field Details

`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` *the transfer-id of the first transfer id the sender will send*

See 2.5.6 Session Flow Control.

`incoming-window` *the initial incoming-window of the sender*

See 2.5.6 Session Flow Control.

`outgoing-window` *the initial outgoing-window of the sender*

See 2.5.6 Session Flow Control.

`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*

A list of commonly defined session capabilities and their meanings can be found here: `http://www.amqp.org/specification/1.0/session-capabilities`.

`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.
A list of commonly defined session properties and their meanings can be found here: `http://www.amqp.org/specification/1.0/session-properties`.

2.7.3 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="name" type="string" mandatory="true"/>
    <field name="handle" type="handle" mandatory="true"/>
    <field name="role" type="role" mandatory="true"/>
    <field name="snd-settle-mode" type="sender-settle-mode" default="mixed"/>
    <field name="rcv-settle-mode" type="receiver-settle-mode" default="first"/>
    <field name="source" type="*" requires="source"/>
    <field name="target" type="*" requires="target"/>
    <field name="unsettled" type="map"/>
    <field name="incomplete-unsettled" type="boolean" default="false"/>
    <field name="initial-delivery-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

`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 (ordered) tuple *(A,B,<name>)*.

`handle`

The handle MUST NOT be used for other open Links. An attempt to attach using a handle which is already associated with a Link MUST be responded to with an immediate `close` carrying a Handle-in-use `session-error`.
To make it easier to monitor AMQP link attach frames, it is recommended that implementations always assign the lowest available handle to this field.

`role` *role of the link endpoint*

`snd-settle-mode` *settlement mode for the Sender*

Determines the settlement policy for deliveries sent at the Sender. When set at the Receiver this indicates the desired value for the settlement mode at the Sender. When set at the Sender this indicates the actual settlement mode in use.

`rcv-settle-mode` *the settlement mode of the Receiver*

Determines the settlement policy for unsettled deliveries received at the Receiver. When set at the Sender this indicates the desired value for the settlement mode at the Receiver. When set at the Receiver this indicates the actual settlement mode in use.

`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 delivery state*

This is used to indicate any unsettled delivery states when a suspended link is resumed. The map is keyed by delivery-tag with values indicating the delivery state. The local and remote delivery states for a given delivery-tag MUST be compared to resolve any in-doubt deliveries. If necessary, deliveries MAY be resent, or resumed based on the outcome of this comparison. See 2.6.13 Resuming Deliveries.
If the local unsettled map is too large to be encoded within a frame of the agreed maximum frame size then the session may be ended with the frame-size-too-small error (see `amqp-error`). The endpoint SHOULD make use of the ability to send an incomplete unsettled map (see below) to avoid sending an error.
The unsettled map MUST NOT contain null valued keys.
When reattaching (as opposed to resuming), the unsettled map MUST be null.

`incomplete-unsettled`

If set to true this field indicates that the unsettled map provided is not complete. When the map is incomplete the recipient of the map cannot take the absence of a delivery tag from the map as evidence of settlement. On receipt of an incomplete unsettled map a sending endpoint MUST NOT send any new deliveries (i.e. deliveries where resume is not set to true) to its partner (and a receiving endpoint which sent an incomplete unsettled map MUST detach with an error on receiving a transfer which does not have the resume flag set to true).

`initial-delivery-count`

This MUST NOT be null if role is sender, and it is ignored if the role is receiver. See 2.6.7 Flow Control.

`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 deliver 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*

A list of commonly defined session capabilities and their meanings can be found here: `http://www.amqp.org/specification/1.0/link-capabilities`.

`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.
A list of commonly defined link properties and their meanings can be found here: `http://www.amqp.org/specification/1.0/link-properties`

2.7.4 Flow

Update link state.

```
<type name="flow" class="composite" source="list" provides="frame">
    <descriptor name="amqp:flow:list" code="0x00000000:0x00000013"/>
    <field name="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"/>
    <field name="delivery-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"/>
    <field name="properties" type="fields"/>
</type>
```

Updates the flow state for the specified Link.

Field Details

`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 2.5.6 Session Flow Control for more details.

`incoming-window`

Defines the maximum number of incoming `transfer` frames that the endpoint can currently receive. See 2.5.6 Session Flow Control for more details.

`next-outgoing-id`

The transfer-id that will be assigned to the next outgoing `transfer` frame. See 2.5.6 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 2.5.6 Session Flow Control for more details.

`handle`

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.
If set to a handle that is not currently associated with an attached Link, the recipient MUST respond by ending the session with an `unattached-handle` session error.

`delivery-count` *the endpoint's delivery-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 delivery-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 2.6.7 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 2.6.7 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 2.6.7 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 2.6.7 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*

`properties` *link state properties*

A list of commonly defined link state properties and their meanings can be found here: `http://www.amqp.org/specification/1.0/link-state-properties`

2.7.5 Transfer

Transfer a Message.

```
<type name="transfer" class="composite" source="list" provides="frame">
    <descriptor name="amqp:transfer:list" code="0x00000000:0x00000014"/>
    <field name="handle" type="handle" mandatory="true"/>
    <field name="delivery-id" type="delivery-number"/>
    <field name="delivery-tag" type="delivery-tag"/>
    <field name="message-format" type="message-format"/>
    <field name="settled" type="boolean"/>
    <field name="more" type="boolean" default="false"/>
    <field name="rcv-settle-mode" type="receiver-settle-mode"/>
    <field name="state" type="*" requires="delivery-state"/>
    <field name="resume" type="boolean" default="false"/>
    <field name="aborted" type="boolean" default="false"/>
    <field name="batchable" type="boolean" default="false"/>
</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

`handle`

Specifies the Link on which the Message is transferred.

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

The delivery-id MUST be supplied on the first transfer of a multi-transfer delivery. On continuation transfers the delivery-id MAY be omitted. It is an error if the delivery-id on a continuation transfer differs from the delivery-id on the first transfer of a delivery.

`delivery-tag`

Uniquely identifies the delivery attempt for a given Message on this Link. This field MUST be specified for the first transfer of a multi transfer message and may only be omitted for continuation transfers.

`message-format` *indicates the message format*

This field MUST be specified for the first transfer of a multi transfer message and may only be omitted for continuation transfers.

`settled`

If not set on the first (or only) transfer for a delivery, then the settled flag MUST be interpreted as being false. For subsequent transfers if the settled flag is left unset then it MUST be interpreted as true if and only if the value of the settled flag on any of the preceding transfers was true; if no preceding transfer was sent with settled being true then the value when unset MUST be taken as false.
If the negotiated value for snd-settle-mode at attachment is `settled`, then this field MUST be true on at least one transfer frame for a delivery (i.e. the delivery must be settled at the Sender at the point the delivery has been completely transferred).
If the negotiated value for snd-settle-mode at attachment is `unsettled`, then this field MUST be false (or unset) on every transfer frame for a delivery (unless the delivery is aborted).

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

Note that if both the more and aborted fields are set to true, the aborted flag takes precedence. That is a receiver should ignore the value of the more field if the transfer is marked as aborted. A sender SHOULD NOT set the more flag to true if it also sets the aborted flag to true.

`rcv-settle-mode`

If `first`, this indicates that the Receiver MUST settle the delivery once it has arrived without waiting for the Sender to settle first.
If `second`, this indicates that the Receiver MUST NOT settle until sending its disposition to the Sender and receiving a settled disposition from the sender.
If not set, this value is defaulted to the value negotiated on link attach.
If the negotiated link value is `first`, then it is illegal to set this field to `second`.
If the message is being sent settled by the Sender, the value of this field is ignored.
The (implicit or explicit) value of this field does not form part of the transfer state, and is not retained if a link is suspended and subsequently resumed.

`state` *the state of the delivery at the sender*

When set this informs the receiver of the state of the delivery at the sender. This is particularly useful when transfers of unsettled deliveries are resumed after a resuming a link. Setting the state on the transfer can be thought of as being equivalent to sending a disposition immediately before the `transfer` performative, i.e. it is the state of the delivery (not the transfer) that existed at the point the frame was sent.
Note that if the `transfer` performative (or an earlier `disposition` performative referring to the delivery) indicates that the delivery has attained a terminal state, then no future `transfer` or `disposition` sent by the sender can alter that terminal state.

`resume` *indicates a resumed delivery*

If true, the resume flag indicates that the transfer is being used to reassociate an unsettled delivery from a dissociated link endpoint. See 2.6.13 Resuming Deliveries for more details.
The receiver MUST ignore resumed deliveries that are not in its local unsettled map. The sender MUST NOT send resumed transfers for deliveries not in its local unsettled map.
If a resumed delivery spans more than one transfer performative, then the resume flag MUST be set to true on the first transfer of the resumed delivery. For subsequent transfers for the same delivery the resume flag may be set to true, or may be omitted.
In the case where the exchange of unsettled maps makes clear that all message data has been successfully transferred to the receiver, and that only the final state (and potentially settlement) at the sender needs to be conveyed, then a resumed delivery may carry no payload and instead act solely as a vehicle for carrying the terminal state of the delivery at the sender.

`aborted` *indicates that the Message is aborted*

Aborted Messages should be discarded by the recipient (any payload within the frame carrying the performative MUST be ignored). An aborted Message is implicitly settled.

`batchable` *batchable hint*

If true, then the issuer is hinting that there is no need for the peer to urgently communicate updated delivery state. This hint may be used to artificially increase the amount of batching an implementation uses when communicating delivery states, and thereby save bandwidth.
If the message being delivered is too large to fit within a single frame, then the setting of batchable to true on any of the `transfer` performatives for the delivery is equivalent to setting batchable to true for all the `transfer` performatives for the delivery.
The batchable value does not form part of the transfer state, and is not retained if a link is suspended and subsequently resumed.

2.7.6 Disposition

Inform remote peer of delivery state changes.

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

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

Note that it is possible for a disposition sent from sender to receiver to refer to a delivery which has not yet completed (i.e. a delivery which is spread over multiple frames and not all frames have yet been sent). The use of such interleaving is discouraged in favor of carrying the modified state on the next `transfer` performative for the delivery.

The disposition performative may refer to deliveries on links that are no longer attached. As long as the links have not been closed or detached with an error then the deliveries are still ”live” and the updated state MUST be applied.

Field Details

`role` *directionality of disposition*

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

`first` *lower bound of deliveries*

Identifies the lower bound of delivery-ids for the deliveries in this set.

`last` *upper bound of deliveries*

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

`settled` *indicates deliveries are settled*

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

`state` *indicates state of deliveries*

Communicates the state of all the deliveries referenced by this disposition.

`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 delivery states. This hint may be used to artificially increase the amount of batching an implementation uses when communicating delivery states, and thereby save bandwidth.

2.7.7 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="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

`handle` *the local handle of the link to be detached*

`closed` *if true then the sender has closed the link*

See 2.6.6 Closing a Link.

`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.

2.7.8 End

End the Session.

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

Indicates that the Session has ended.

Field Details

`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.

2.7.9 Close

Signal a Connection close.

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

`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.

2.8 Definitions

2.8.1 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	sender
true	receiver

2.8.2 Sender Settle Mode

Settlement policy for a Sender.

```
<type name="sender-settle-mode" class="restricted" source="ubyte">
    <choice name="unsettled" value="0"/>
    <choice name="settled" value="1"/>
    <choice name="mixed" value="2"/>
</type>
```

Valid Values

0	The Sender will send all deliveries initially unsettled to the Receiver.
1	The Sender will send all deliveries settled to the Receiver.
2	The Sender may send a mixture of settled and unsettled deliveries to the Receiver.

2.8.3 Receiver Settle Mode

Settlement policy for a Receiver.

```
<type name="receiver-settle-mode" class="restricted" source="ubyte">
    <choice name="first" value="0"/>
    <choice name="second" value="1"/>
</type>
```

Valid Values

0	The Receiver will spontaneously settle all incoming transfers.
1	The Receiver will only settle after sending the `disposition` to the Sender and receiving a `disposition` indicating settlement of the delivery from the sender.

2.8.4 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.

2.8.5 Seconds

A duration measured in seconds.

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

2.8.6 Milliseconds

A duration measured in milliseconds.

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

2.8.7 Delivery Tag

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

A delivery-tag may be up to 32 octets of binary data.

2.8.8 Delivery Number

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

2.8.9 Transfer Number

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

2.8.10 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.

2.8.11 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
```

2.8.12 IETF Language Tag

An IETF language tag as defined by BCP 47.

```
<type name="ietf-language-tag" class="restricted" source="symbol"/>
```

IETF language tags are abbreviated language codes as defined in the IETF Best Current Practice BCP-47 (http://www.rfc-editor.org/rfc/bcp/bcp47.txt) (incorporating RFC-5646 (http://www.rfc-editor.org/rfc/rfc5646.txt)). A list of registered subtags is maintained in the IANA Language Subtag Registry (http://www.iana.org/assignments/language-subtag-registry).

All AMQP implementations should understand at the least the IETF language tag *en-US* (note that this uses a hyphen separator, not an underscore).

2.8.13 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 (this excludes the possibility of a null key). There is no further restriction implied by the *fields* type on the allowed values for the entries or the set of allowed keys.

2.8.14 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-condition" mandatory="true"/>
    <field name="description" type="string"/>
    <field name="info" type="fields"/>
</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*

2.8.15 AMQP Error

Shared error conditions.

```
<type name="amqp-error" class="restricted" source="symbol" provides="error-condition">
    <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"/>
    <choice name="frame-size-too-small" value="amqp:frame-size-too-small"/>
</type>
```

Valid Values

amqp:internal-error

An internal error occurred. 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.

amqp:frame-size-too-small

The peer cannot send a frame because the smallest encoding of the performative with the currently valid values would be too large to fit within a frame of the agreed maximum frame size. When transferring a message the message data can be sent in multiple `transfer` frames thereby avoiding this error. Similarly when attaching a link with a large unsettled map the endpoint may make use of the incomplete-unsettled flag to avoid the need for overly large frames.

2.8.16 Connection Error

Symbols used to indicate connection error conditions.

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

amqp:connection:redirect

The container is no longer available on the current connection. The peer should attempt reconnection to the container using the details provided in the info map.

hostname	the hostname of the container. This is the value that should be supplied in the *hostname* field of the `open` frame, and suring the SASL and TLS negotiation (if used).
network-host	the DNS hostname or IP address of the machine hosting the container.
port	the port number on the machine hosting the container.

2.8.17 Session Error

Symbols used to indicate session error conditions.

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

amqp:session:handle-in-use

An attach was received using a handle that is already in use for an attached Link.

amqp:session:unattached-handle

A frame (other than attach) was received referencing a handle which is not currently in use of an attached Link.

2.8.18 Link Error

Symbols used to indicate link error conditions.

```
<type name="link-error" class="restricted" source="symbol" provides="error-condition">
    <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"/>
    <choice name="redirect" value="amqp:link:redirect"/>
    <choice name="stolen" value="amqp:link:stolen"/>
</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 link.

amqp:link:message-size-exceeded

The peer sent a larger message than is supported on the link.

amqp:link:redirect

The address provided cannot be resolved to a terminus at the current container. The info map may contain the following information to allow the client to locate the attach to the terminus.

hostname	the hostname of the container hosting the terminus. This is the value that should be supplied in the *hostname* field of the `open` frame, and during SASL and TLS negotiation (if used).
network-host	the DNS hostname or IP address of the machine hosting the container.
port	the port number on the machine hosting the container.
address	the address of the terminus at the container.

amqp:link:stolen

The link has been attached elsewhere, causing the existing attachment to be forcibly closed.

2.8.19 Constant Definitions

PORT	5672	the IANA assigned port number for AMQP. The standard AMQP port number that has been assigned by IANA for TCP, UDP, and SCTP. There are currently no UDP or SCTP mappings defined for AMQP. The port number is reserved for future transport mappings to these protocols.
SECURE-PORT	5671	the IANA assigned port number for secure AMQP (amqps). 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.
MAJOR	1	major protocol version.
MINOR	0	minor protocol version.
REVISION	0	protocol revision.
MIN-MAX-FRAME-SIZE	512	the lower bound for the agreed maximum frame size (in bytes). 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 3

Messaging

3.1 Introduction

The messaging layer builds on top of the concepts described in books I and II. 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
- delivery 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

3.2 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 sections for annotation at the head and tail 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* consists of sections standard properties, application-properties, and application-data (the body).

```
                                                     Bare Message
                                                          |
                                    .---------------------+--------------------.
                                    |                                          |
+--------+-------------+-------------+------------+--------------+--------------+--------+
| header | delivery-   | message-    | properties | application- | application- | footer |
|        | annotations | annotations |            | properties   | data         |        |
+--------+-------------+-------------+------------+--------------+--------------+--------+
|                                                                                        |
'-------------------------------------------+--------------------------------------------'
                                            |
                                    Annotated Message
```

The bare message is immutable within the AMQP network. That is none of the sections can be changed by any node acting as an AMQP intermediary. If a section of the bare message is omitted, one may not be inserted by an intermediary. The exact encoding of sections of the bare message MUST NOT be modified. This preserves message hashes, HMACs and signatures based on the binary encoding of the bare message.

The exact structure of a message, together with its encoding, is defined by the message format. This document defines the structure and semantics of message format 0 (MESSAGE-FORMAT). Altogether a message consists of the following sections:

- Zero or one `header`.
- Zero or one `delivery-annotations`.
- Zero or one `message-annotations`.
- Zero or one `properties`.
- Zero or one `application-properties`.
- The body consists of either: one or more `data` sections, one or more `amqp-sequence` sections, or a single `amqp-value` section.
- Zero or one `footer`.

3.2.1 Header

Transport headers for a Message.

```
<type name="header" class="composite" source="list" provides="section">
    <descriptor name="amqp:header:list" code="0x00000000:0x00000070"/>
    <field name="durable" type="boolean"/>
    <field name="priority" type="ubyte"/>
    <field name="ttl" type="milliseconds"/>
    <field name="first-acquirer" type="boolean"/>
    <field name="delivery-count" type="uint"/>
</type>
```

The header section carries standard delivery details about the transfer of a Message through the AMQP network. If the header section is omitted the receiver MUST assume the appropriate default values for the fields within the `header` unless other target or node specific defaults have otherwise been set.

Field Details

`durable` *specify durability requirements*

Durable Messages MUST NOT be lost even if an intermediary is unexpectedly terminated and restarted. A target which is not capable of fulfilling this guarantee MUST NOT accept messages where the durable header is set to true: if the source allows the `rejected` outcome then the message should be rejected with the `precondition-failed` error, otherwise the link must be detached by the receiver with the same error.

`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.

`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 time of arrival at an intermediary. Messages that live longer than their expiration time will be discarded (or dead lettered). When a message is transmitted by an intermediary that was received with a ttl, the transmitted message's header should contain a ttl that is computed as the difference between the current time and the formerly computed message expiration time, i.e. the reduced ttl, so that messages will eventually die if they end up in a delivery loop.

`first-acquirer`

If this value is true, then this message has not been acquired by any other Link. If this value is false, then this message may have previously been acquired by another Link or Links.

`delivery-count` *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 delivery may be a duplicate. On first delivery, the value is zero. It is incremented upon an outcome being settled at the sender, according to rules defined for each outcome.

3.2.2 Delivery Annotations

```
<type name="delivery-annotations" class="restricted" source="annotations" provides="section">
    <descriptor name="amqp:delivery-annotations:map" code="0x00000000:0x00000071"/>
</type>
```

The delivery-annotations section is used for delivery-specific non-standard properties at the head of the message. Delivery annotations convey information from the sending peer to the receiving peer. If the recipient does not understand the annotation it cannot be acted upon and its effects (such as any implied propagation) cannot be acted upon. Annotations may be specific to one implementation, or common to multiple implementations. The capabilities negotiated on link attach and on the `source` and `target` should be used to establish which annotations a peer supports. A registry of defined annotations and their meanings can be found here: `http://www.amqp.org/specification/1.0/delivery-annotations`.

If the delivery-annotations section is omitted, it is equivalent to a delivery-annotations section containing an empty map of annotations.

3.2.3 Message Annotations

```
<type name="message-annotations" class="restricted" source="annotations" provides="section">
    <descriptor name="amqp:message-annotations:map" code="0x00000000:0x00000072"/>
</type>
```

The message-annotations section is used for properties of the message which are aimed at the infrastructure and should be propagated across every delivery step. Message annotations convey information about the message. Intermediaries MUST propagate the annotations unless the annotations are explicitly augmented or modified (e.g. by the use of the `modified` outcome).

The capabilities negotiated on link attach and on the `source` and `target` may be used to establish which annotations a peer understands, however it a network of AMQP intermediaries it may not be possible to know if every intermediary will understand the annotation. Note that for some annotation it may not be necessary for the intermediary to understand their purpose - they may be being used purely as an attribute which can be filtered on.

A registry of defined annotations and their meanings can be found here: `http://www.amqp.org/specification/1.0/message-annotations`.

If the message-annotations section is omitted, it is equivalent to a message-annotations section containing an empty map of annotations.

3.2.4 Properties

Immutable properties of the Message.

```
<type name="properties" class="composite" source="list" provides="section">
    <descriptor name="amqp:properties:list" code="0x00000000:0x00000073"/>
    <field name="message-id" type="*" requires="message-id"/>
    <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="*" requires="message-id"/>
    <field name="content-type" type="symbol"/>
    <field name="content-encoding" type="symbol"/>
    <field name="absolute-expiry-time" type="timestamp"/>
    <field name="creation-time" type="timestamp"/>
    <field name="group-id" type="string"/>
    <field name="group-sequence" type="sequence-no"/>
    <field name="reply-to-group-id" type="string"/>
</type>
```

The properties section is used for a defined set of standard properties of the message. The properties section is part of the bare message and thus must, if retransmitted by an intermediary, remain completely unaltered.

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. A broker 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.

`content-type` *MIME content type*

The RFC-2046 MIME type for the Message's application-data section (body).
As per RFC-2046 this may contain a charset parameter defining the character encoding used: e.g. 'text/plain; charset="utf-8"'. For clarity, the correct MIME type for a truly opaque binary section is application/octet-stream.
When using an application-data section with a section code other than *data*, content-type, if set, SHOULD be set to a MIME type of message/x-amqp+?, where '?' is either data, map or list.

`content-encoding` *MIME content type*

The Content-Encoding property is used as a modifier to the content-type. When present, its value indicates what additional content encodings have been applied to the application-data, and thus what decoding mechanisms must be applied in order to obtain the media-type referenced by the content-type header field.
Content-Encoding is primarily used to allow a document to be compressed without losing the identity of its underlying content type.
Content Encodings are to be interpreted as per Section 3.5 of RFC 2616. Valid Content Encodings are registered at IANA as "Hypertext Transfer Protocol (HTTP) Parameters" (http://www.iana.org/assignments/http-parameters/http-parameters.xml).
Content-Encoding MUST not be set when the application-data section is other than *data*.
Implementations MUST NOT use the *identity* encoding. Instead, implementations should not set this property. Implementations SHOULD NOT use the *compress* encoding, except as to remain compatible with messages originally sent with other protocols, e.g. HTTP or SMTP.
Implementations SHOULD NOT specify multiple content encoding values except as to be compatible with messages originally sent with other protocols, e.g. HTTP or SMTP.

`absolute-expiry-time` *the time when this message is considered expired*

An absolute time when this message is considered to be expired.

`creation-time` *the time when this message was created*

An absolute time when this message was created.

`group-id` *the group this message belongs to*

Identifies the group the message belongs to.

`group-sequence` *the sequence-no of this message within its group*

The relative position of this message within its group.

`reply-to-group-id` *the group the reply message belongs to*

This is a client-specific id that is used so that client can send replies to this message to a specific group.

3.2.5 Application Properties

```
<type name="application-properties" class="restricted" source="map" provides="section">
    <descriptor name="amqp:application-properties:map" code="0x00000000:0x00000074"/>
</type>
```

The application-properties section is a part of the bare message used for structured application data. Intermediaries may use the data within this structure for the purposes of filtering or routing.

The keys of this map are restricted to be of type `string` (which excludes the possibility of a null key) and the values are restricted to be of simple types only, that is (excluding `map`, `list`, and `array` types).

3.2.6 Data

```
<type name="data" class="restricted" source="binary" provides="section">
    <descriptor name="amqp:data:binary" code="0x00000000:0x00000075"/>
</type>
```

A data section contains opaque binary data.

3.2.7 AMQP Sequence

```
<type name="amqp-sequence" class="restricted" source="list" provides="section">
    <descriptor name="amqp:amqp-sequence:list" code="0x00000000:0x00000076"/>
</type>
```

A sequence section contains an arbitrary number of structured data elements.

3.2.8 AMQP Value

```
<type name="amqp-value" class="restricted" source="*" provides="section">
    <descriptor name="amqp:amqp-value:*" code="0x00000000:0x00000077"/>
</type>
```

An amqp-value section contains a single AMQP value.

3.2.9 Footer

Transport footers for a Message.

```
<type name="footer" class="restricted" source="annotations" provides="section">
    <descriptor name="amqp:footer:map" code="0x00000000:0x00000078"/>
</type>
```

The footer section is used for details about the message or delivery which can only be calculated or evaluated once the whole bare message has been constructed or seen (for example message hashes, HMACs, signatures and encryption details).

A registry of defined footers and their meanings can be found here: `http://www.amqp.org/specification/1.0/footer`.

3.2.10 Annotations

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

The *annotations* type is a map where the keys are restricted to be of type `symbol` or of type `ulong`. All ulong keys, and all symbolic keys except those beginning with ”x-” are reserved. On receiving an annotations map containing keys or values which it does not recognize, and for which the key does not begin with the string ”x-opt-” an AMQP container MUST detach the link with the not-implemented amqp-error.

3.2.11 Message ID ULong

```
<type name="message-id-ulong" class="restricted" source="ulong" provides="message-id"/>
```

3.2.12 Message ID UUID

```
<type name="message-id-uuid" class="restricted" source="uuid" provides="message-id"/>
```

3.2.13 Message ID Binary

```
<type name="message-id-binary" class="restricted" source="binary" provides="message-id"/>
```

3.2.14 Message ID String

```
<type name="message-id-string" class="restricted" source="string" provides="message-id"/>
```

3.2.15 Address String

Address of a Node.

```
<type name="address-string" class="restricted" source="string" provides="address"/>
```

3.2.16 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.

3.3 Distribution Nodes

3.3.1 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 delivery 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 Message is ineligible for *acquiring* transfers to any other Links.

A Message will remain ACQUIRED at the distribution node until the transfer is settled. The delivery state at the receiver determines how the message transitions when the transfer is settled. If the delivery 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 transitioned to a terminal outcome at the distribution node regardless of receiver state.

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

Figure 3.1: Message State Transitions

3.4 Delivery State

The Messaging layer defines a concrete set of delivery states which can be used (via the `disposition` frame) to indicate the state of the message at the receiver. Delivery states may be either terminal or non-terminal. Once a delivery reaches a terminal delivery-state, the state for that delivery will no longer change. A terminal delivery-state is referred to as an *outcome*.

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

The following non-terminal delivery-state is formally defined by the messaging layer for use during link recovery to allow the sender to resume the transfer of a large message without retransmitting all the message data:

- `received`: indicates partial message data seen by the receiver as well as the starting point for a resumed transfer

3.4.1 Received

```
<type name="received" class="composite" source="list" provides="delivery-state">
    <descriptor name="amqp:received:list" code="0x00000000:0x00000023"/>
    <field name="section-number" type="uint" mandatory="true"/>
    <field name="section-offset" type="ulong" mandatory="true"/>
</type>
```

At the target the `received` state indicates the furthest point in the payload of the message which the target will not need to have resent if the link is resumed. At the source the `received` state represents the earliest point in the payload which the Sender is able to resume transferring at in the case of link resumption. When resuming a delivery, if this state is set on the first `transfer` performative it indicates the offset in the payload at which the first resumed delivery is starting. The Sender MUST NOT send the `received` state on `transfer` or `disposition` performatives except on the first `transfer` performative on a resumed delivery.

Field Details

`section-number`

When sent by the Sender this indicates the first section of the message (with section-number 0 being the first section) for which data can be resent. Data from sections prior to the given section cannot be retransmitted for this delivery.
When sent by the Receiver this indicates the first section of the message for which all data may not yet have been received.

`section-offset`

When sent by the Sender this indicates the first byte of the encoded section data of the section given by section-number for which data can be resent (with section-offset 0 being the first byte). Bytes from the same section prior to the given offset section cannot be retransmitted for this delivery.
When sent by the Receiver this indicates the first byte of the given section which has not yet been received. Note that if a receiver has received all of section number X (which contains N bytes of data), but none of section number X + 1, then it may indicate this by sending either Received(section-number=X, section-offset=N) or Received(section-number=X+1, section-offset=0). The state Received(section-number=0, section-offset=0) indicates that no message data at all has been transferred.

3.4.2 Accepted

The accepted outcome.

```
<type name="accepted" class="composite" source="list" provides="delivery-state, outcome">
    <descriptor name="amqp:accepted:list" code="0x00000000:0x00000024"/>
</type>
```

At the source the accepted state means that the message has been retired from the node, and transfer of payload data will not be able to be resumed if the link becomes suspended. A delivery may become accepted at the source even before all transfer frames have been sent, this does not imply that the remaining transfers for the delivery will not be sent - only the aborted flag on the `transfer` performative can be used to indicate a premature termination of the transfer.

At the target, the accepted outcome is used to indicate that an incoming Message has been successfully processed, and that the receiver of the Message is expecting the sender to transition the delivery to the accepted state at the source.

The accepted outcome does not increment the *delivery-count* in the header of the accepted Message.

3.4.3 Rejected

The rejected outcome.

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

At the target, 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-count* to be incremented in the header of the rejected Message.

At the source, the rejected outcome means that the target has informed the source that the message was rejected, and the source has taken the required action. The delivery SHOULD NOT ever spontaneously attain the rejected state at the source.

Field Details

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

The value supplied in this field will be placed in the delivery-annotations of the rejected Message associated with the symbolic key ”rejected”.

3.4.4 Released

The released outcome.

```
<type name="released" class="composite" source="list" provides="delivery-state, outcome">
    <descriptor name="amqp:released:list" code="0x00000000:0x00000026"/>
</type>
```

At the source the released outcome means that the message is no longer acquired by the receiver, and has been made available for (re-)delivery to the same or other targets receiving from the node. The message is unchanged at the node (i.e. the *delivery-count* of the header of the released Message MUST NOT be incremented). As released is a terminal outcome, transfer of payload data will not be able to be resumed if the link becomes suspended. A delivery may become released at the source even before all transfer frames have been sent, this does not imply that the remaining transfers for the delivery will not be sent. The source MAY spontaneously attain the released outcome for a Message (for example the source may implement some sort of time bound acquisition lock, after which the acquisition of a message at a node is revoked to allow for delivery to an alternative consumer).

At the target, the released outcome is used to indicate that a given transfer was not and will not be acted upon.

3.4.5 Modified

The modified outcome.

```
<type name="modified" class="composite" source="list" provides="delivery-state, outcome">
    <descriptor name="amqp:modified:list" code="0x00000000:0x00000027"/>
    <field name="delivery-failed" type="boolean"/>
    <field name="undeliverable-here" type="boolean"/>
    <field name="message-annotations" type="fields"/>
</type>
```

At the source the modified outcome means that the message is no longer acquired by the receiver, and has been made available for (re-)delivery to the same or other targets receiving from the node. The message has been changed at the node in the ways indicated by the fields of the outcome. As modified is a terminal outcome, transfer of payload data will not be able to be resumed if the link becomes suspended. A delivery may become modified at the source even before all transfer frames have been sent, this does not imply that the remaining transfers for the delivery will not be sent. The source MAY spontaneously attain the modified outcome for a Message (for example the source may implement some sort of time bound acquisition lock, after which the acquisition of a message at a node is revoked to allow for delivery to an alternative consumer with the message modified in some way to denote the previous failed, e.g. with delivery-failed set to true).

At the target, the modified outcome is used to indicate that a given transfer was not and will not be acted upon, and that the message should be modified in the specified ways at the node.

Field Details

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

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

`undeliverable-here` *prevent redelivery*

If the undeliverable-here is set, then any Messages released MUST NOT be redelivered to the modifying Link Endpoint.

`message-annotations` *message attributes*

Map containing attributes to combine with the existing *message-annotations* held in the Message's header section. Where the existing message-annotations 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-annotations has no such value, the value in this map is added.

3.4.6 Resuming Deliveries Using Delivery States

In 2.6.13 Resuming Deliveries the general scheme for how two endpoints should re-establish state after link resumption was provided. The concrete delivery states defined above allow for a more comprehensive set of examples of link resumption.

```
Peer                                  Partner
=======================================================================

ATTACH(name=N, handle=1,   --+     +-- ATTACH(name=N, handle=2,
       role=sender,            \   /          role=receiver,
       source=X,                \ /           source=X,
       target=Y,                 x            target=Y,
       unsettled=               / \           unsettled=
        { 1 -> null,           /   \           { 2 -> Received(3,0),
          2 -> null,       <-+     +->           3 -> Accepted,
          3 -> null,                             4 -> null,
          4 -> null,                             6 -> Received(2,0),
          5 -> Received(0,200),                  7 -> Received(0,100),
          6 -> Received(1,150),                  8 -> Accepted,
          7 -> Received(0,500),                  9 -> null,
          8 -> Received(3,5),                   11 -> Received(1,2000),
          9 -> Received(2,0),                   12 -> Accepted,
         10 -> Accepted,                        13 -> Released,
         11 -> Accepted,                        14 -> null }
         12 -> Accepted,
         13 -> Accepted,
         14 -> Accepted }

-----------------------------------------------------------------------
Key:

Received(x,y) means Received(section-number=x, section-offset=y)
```

In this example, for delivery-tags 1 to 4 inclusive the sender indicates that it can resume sending from the start of the message.

For delivery-tag 1, the receiver has no record of the delivery. To preserve "at least once", or "at most once" delivery guarantees, the sender MUST resend the message, however the delivery is not being resumed (since the receiver does not remember the delivery tag) so transfers MUST NOT have the resume flag set to true. If the sender were to mark the transfers as resumes then they would be ignored at the receiver.

For delivery-tag 2, the receiver has retained some of the data making up the message, but not the whole. In order to complete the delivery the sender must resume sending from some point before or at the next position which the receiver is expecting.

```
TRANSFER(delivery-id=1,    ----------> ** Append message data not     **
         delivery-tag=2,               ** seen previously to delivery **
    (1)  state=Received(3,0),          ** state.                      **
         resume=true)
{ ** payload ** }


(1) state could be
    a) null, meaning that the transfer is being resumed from the first
       byte of section number 0 (and the receiver MUST ignore all data
       up to the first position it has not previously received).
    b) Received with section number 0, 1 or 2 and an offset, indicating
       that the payload data on the first frame of the resumed delivery
       starts at the given point, and that the receiver MUST ignore all
       data up to the first position it has not previously received.
    c) Received(3,0) indicating that the resumption will start at the
       first point which the receiver has not previously received.
```

For delivery-tag 3, the receiver indicates that it has processed the delivery to the point where it desires a terminal outcome (in this case `accepted`). In this case the sender will either apply that outcome at the source, or in the rare case that it cannot apply that outcome, indicate the terminal outcome that has been applied. To do this the sender MUST send a resuming transfer to associate delivery-tag 3 with a delivery-id. On this transfer the sender SHOULD set the delivery-state at the source. If this is the same outcome as at the receiver then the sender MAY also send the resuming transfer as settled.

```
TRANSFER(delivery-id=2,    ----------> ** Processes confirmation that **
         delivery-tag=3,               ** was accepted, and settles.  **
         settled=true,
         more=false,
         state=Accepted,
         resume=true)
```

For delivery-tag 4, the receiver indicates that it is aware that the delivery had begun, but does not provide any indication that it has retained any data about that delivery except the fact of its existence. To preserve "at least once" or "at most once" delivery guarantees, the sender MUST resend the whole message. Unlike the case with delivery-tag 1 the resent delivery MUST be sent with the resume flag set to true and the delivery-tag set to 4. (While this use of null in the receivers map is valid, it is discouraged. It is recommended that receiver SHOULD NOT retain such an entry in its map, in which case the situation would be as for delivery-tag 1 in this example).

```
TRANSFER(delivery-id=3,    ----------> ** Processes in the same way   **
         delivery-tag=4,               ** as we be done for a non-    **
    (1)  state=null,                   ** resumed delivery.           **
         resume=true)
{ ** payload ** }

(1) Alternatively (and equivalently) state could be
    Received(section-number=0, section-offset=0)
```

For delivery-tags 5 to 9 inclusive the sender indicates that it can resume at some point beyond start of the message data. This is usually indicative of the fact that the receiver had previously confirmed reception of message data to the given point, removing responsibility from the sender to retain the ability to resend that data after resuming the link. The sender MAY still retain the ability to resend the message as a new delivery (i.e. it MAY not have completely discarded the data from which the original delivery was generated).

For delivery-tag 5, the receiver has no record of the delivery. To preserve "at least once", or "at most once" delivery guarantees, the sender MUST resend the message, however the delivery is not being resumed (since the receiver does not remember the delivery tag) so transfers MUST NOT have the resume flag set to true. If the sender does not enough data to resend the message, then he sender MAY take some action to indicate that it believes there is a possibility that there has been message loss.

For delivery-tag 6, the receiver has retained some of the data making up the message, but not the whole. The first position within the message which the receiver has not received is after the first position at which the sender can resume sending. In order to complete the delivery the sender must resume sending from some point before or at the next position which the receiver is expecting.

```
TRANSFER(delivery-id=4,    ----------> ** Append message data not     **
         delivery-tag=6,               ** seen previously to delivery **
    (1)  state=Received(2,0),          ** state.                      **
         resume=true)
{ ** payload ** }

(1) state could be any point between Received(1,150) and Received(2,0)
    inclusive. The receiver MUST ignore all data up to the first
    position it has not previously received (i.e. section 2 offset 0).
```

For delivery-tag 7, the receiver has retained some of the data making up the message, but not the whole. The first position within the message which the receiver has not received is before the first position at which the sender can resume sending. It is thus not possible for the sender to resume sending the message to completion. The only option available to the sender is to abort the transfer and to then (if possible) resend as a new delivery or else to report the possible message loss in some way if it cannot.

```
TRANSFER(delivery-id=5,     ----------> ** Discard any state relating  **
         delivery-tag=7,                ** to the message delivery.    **
         resume=true,
         aborted=true)
```

For delivery-tag 8, the receiver indicates that it has processed the delivery to the point where it desires a terminal outcome (in this case `accepted`). This is the same case as for delivery-tag 3.

```
TRANSFER(delivery-id=6,     ----------> ** Processes confirmation that **
         delivery-tag=8,                ** was accepted, and settles.  **
         settled=true,
         more=false,
         state=Accepted,
         resume=true)
```

For delivery-tag 9, the receiver indicates that it is aware that the delivery had begun, but does not provide any indication that it has retained any data about that delivery except the fact of its existence. This is the same case as for delivery-tag 7.

```
TRANSFER(delivery-id=7,     ----------> ** Discard any state relating  **
         delivery-tag=9,                ** to the message delivery.    **
         resume=true,
         aborted=true)
```

For delivery-tags 10 to 14 inclusive the sender indicates that it has reached a terminal outcome, namely `accepted`. Once the sender has arrived at a terminal outcome it may not change. As such, if a sender is capable of resuming a delivery (even if the only possible outcome of the delivery is a pre-defined terminal outcome such as `accepted`) it MUST NOT use this state as the value of the state in its unsettled map until it is sure that the receiver will not require the resending of the message data.

For delivery-tag 10 the receiver has no record of the delivery. However, in contrast to the cases of delivery-tag 1 and delivery-tag 5, since we know that the sender can only have arrived at this state through knowing that the receiver has received the whole message (or that the sender had spontaneously reached a terminal outcome with no possibility of resumption) we have no need to resend the message.

For delivery-tag 11 we have to assume that the sender spontaneously attained the terminal outcome (and is unable to resume). In this case the sender can simply abort the delivery as it cannot be resumed.

```
TRANSFER(delivery-id=8,     ----------> ** Discard any state relating  **
         delivery-tag=11,               ** to the message delivery.    **
         resume=true,
         aborted=true)
```

For delivery-tag 12 both the sender and receiver have attained the same view of the terminal outcome, but neither has settled. In this case the sender should simply settle the delivery.

```
TRANSFER(delivery-id=9,     ----------> ** Locally settle the delivery **
         delivery-tag=12,
         settled=true,
         resume=true)
```

For delivery-tag 13 the sender and receiver have both attained terminal outcomes, but the outcomes differ. In this case, since the outcome actually takes effect at the sender, it is the sender's view that is definitive. The sender thus MUST restate this as the terminal outcome, and the receiver should then echo this and settle.

```
TRANSFER(delivery-id=10    ----------> ** Update any state affected   **
         delivery-tag=13,              ** by the actual outcome, then **
         settled=false,                ** settle the delivery         **
         state=Accepted
         resume=true)
                           <---------- DISPOSITION(first=10, last=10,
                                                   state=Accepted,
                                                   settled=true)
```

For delivery-tag 14 the case is essentially the same as for delivery-tag 11, as the null state at the receiver is essentially identical to having the state Receivedsection-number=0, section-offset=0.

```
TRANSFER(delivery-id=11,   ----------> ** Discard any state relating  **
         delivery-tag=14,              ** to the message delivery.    **
         resume=true,
         aborted=true)
```

3.5 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 behavior 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.

3.5.1 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 delivery state).

3.5.2 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 behavior 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 delivery 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.

3.5.3 Source

```
<type name="source" class="composite" source="list" provides="source">
    <descriptor name="amqp:source:list" code="0x00000000:0x00000028"/>
    <field name="address" type="*" requires="address"/>
    <field name="durable" type="terminus-durability" default="none"/>
    <field name="expiry-policy" type="terminus-expiry-policy" default="session-end"/>
    <field name="timeout" type="seconds" default="0"/>
    <field name="dynamic" type="boolean" default="false"/>
    <field name="dynamic-node-properties" type="node-properties"/>
    <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>
```

For containers which do not implement address resolution (and do not admit spontaneous link attachment from their partners) but are instead only used as producers of messages, it is unnecessary to provide spurious detail on the source. For this purpose it is possible to use a ”minimal” source in which all the fields are left unset.

Field Details

`address` *the address of the source*

The address of the source MUST NOT be set when sent on a `attach` frame sent by the receiving Link Endpoint where the dynamic flag is set to true (that is where the receiver is requesting the sender to create an addressable node).
The address of the source MUST be set when sent on a `attach` frame sent by the sending Link Endpoint where the dynamic flag is set to true (that is where the sender has created an addressable node at the request of the receiver and is now communicating the address of that created node). The generated name of the address SHOULD include the link name and the container-id of the remote container to allow for ease of identification.

`durable` *indicates the durability of the terminus*

Indicates what state of the terminus will be retained durably: the state of durable messages, only existence and configuration of the terminus, or no state at all.

`expiry-policy` *the expiry policy of the Source*

See subsection 3.5.6 Terminus Expiry Policy.

`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 to true 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 to true by the sending Link Endpoint this field indicates creation of a dynamically created Node. 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.

`dynamic-node-properties` *properties of the dynamically created Node*

If the dynamic field is not set to true this field must be left unset.
When set by the receiving Link endpoint, this field contains the desired properties of the Node the receiver wishes to be created. When set by the sending Link endpoint this field contains the actual properties of the dynamically created node. See subsection 3.5.9 Node Properties.

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

This field MUST be set by the sending end of the Link if the endpoint supports more than one distribution-mode. 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*

See subsection 3.5.8 Filter Set. The receiving endpoint sets its desired filter, the sending endpoint sets the filter actually in place (including any filters defaulted at the node). The receiving endpoint MUST check that the filter in place meets its needs and take responsibility for detaching if it does not.

`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 Source 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 (if the default-outcome is not set, and no outcomes are provided, then the `accepted` outcome must be supported by the source).
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*

See `http://www.amqp.org/specification/1.0/source-capabilities`.

3.5.4 Target

```
<type name="target" class="composite" source="list" provides="target">
    <descriptor name="amqp:target:list" code="0x00000000:0x00000029"/>
    <field name="address" type="*" requires="address"/>
    <field name="durable" type="terminus-durability" default="none"/>
    <field name="expiry-policy" type="terminus-expiry-policy" default="session-end"/>
    <field name="timeout" type="seconds" default="0"/>
    <field name="dynamic" type="boolean" default="false"/>
    <field name="dynamic-node-properties" type="node-properties"/>
    <field name="capabilities" type="symbol" multiple="true"/>
</type>
```

For containers which do not implement address resolution (and do not admit spontaneous link attachment from their partners) but are instead only used as consumers of messages, it is unnecessary to provide spurious detail on the source. For this purpose it is possible to use a ”minimal” target in which all the fields are left unset.

Field Details

`address` *The address of the target.*

The address of the target MUST NOT be set when sent on a `attach` frame sent by the sending Link Endpoint where the dynamic flag is set to true (that is where the sender is requesting the receiver to create an addressable node).
The address of the source MUST be set when sent on a `attach` frame sent by the receiving Link Endpoint where the dynamic flag is set to true (that is where the receiver has created an addressable node at the request of the sender and is now communicating the address of that created node). The generated name of the address SHOULD include the link name and the container-id of the remote container to allow for ease of identification.

`durable` *indicates the durability of the terminus*

Indicates what state of the terminus will be retained durably: the state of durable messages, only existence and configuration of the terminus, or not state at all.

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

See subsection 3.5.6 Terminus Expiry Policy.

`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 to true 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 to true by the receiving Link Endpoint this field indicates creation of a dynamically created Node. 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.

`dynamic-node-properties` *properties of the dynamically created Node*

If the dynamic field is not set to true this field must be left unset.
When set by the sending Link endpoint, this field contains the desired properties of the Node the sender wishes to be created. When set by the receiving Link endpoint this field contains the actual properties of the dynamically created node. See subsection 3.5.9 Node Properties.

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

See `http://www.amqp.org/specification/1.0/target-capabilities`.

3.5.5 Terminus Durability

Durability policy for a Terminus.

```
<type name="terminus-durability" class="restricted" source="uint">
    <choice name="none" value="0"/>
    <choice name="configuration" value="1"/>
    <choice name="unsettled-state" value="2"/>
</type>
```

Determines which state of the Terminus is held durably value.

Valid Values

0	No Terminus state is retained durably.
1	Only the existence and configuration of the Terminus is retained durably.
2	In addition to the existence and configuration of the Terminus, the unsettled state for durable messages is retained durably.

3.5.6 Terminus Expiry Policy

Expiry policy for a Terminus.

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

Determines when the expiry timer of a Terminus starts counting down from the timeout value. If the link is subsequently re-attached before the Terminus is expired, then the count down is aborted. If the conditions for the terminus-expiry-policy are subsequently re-met, the expiry timer restarts from its originally configured timeout value.

Valid Values

link-detach	The expiry timer starts when Terminus is detached.

session-end	The expiry timer starts when the most recently associated session is ended.
connection-close	The expiry timer starts when most recently associated connection is closed.
never	The Terminus never expires.

3.5.7 Standard Distribution 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

3.5.8 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 ether `null` or of a described type which provides the archetype *filter*. A filter acts as a function on a message which returns a boolean result indicating whether the message may pass through that filter or not. A message will pass through a filter-set if and only if it passes through each of the named filters. If the value for a given key is null, this acts as if there were no such key present (i.e. all messages pass through the null filter).

Filter types are a defined extension point. The filter types that a given `source` supports will be indicated by the capabilities of the `source`. Common filter types, along with the capabilities they are associated with are registered here: `http://www.amqp.org/specification/1.0/filters`.

3.5.9 Node Properties

Properties of a Node.

```
<type name="node-properties" class="restricted" source="fields"/>
```

A symbol-keyed map containing properties of a Node - used when requesting creation or reporting the creation of a dynamic Node.

The following common properties are defined:

lifetime-policy	The lifetime of a dynamically generated node.

Definitionally, the lifetime will never be less than the lifetime of the link which caused its creation, however it is possible to extend the lifetime of dynamically created node using a lifetime policy. The value of this entry must be one of the defined *lifetime-policies*: `delete-on-close`, `delete-on-no-links`, `delete-on-no-messages` or `delete-on-no-links-or-messages`.

supported-dist-modes
The distribution modes that the node supports.

The value of this entry must be one or more symbols which are valid *distribution-mode*s. That is the value must be of the same type as would be valid in a field defined as with the following attributes:

type="symbol" multiple="true" requires="distribution-mode"

3.5.10 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"/>
</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.

3.5.11 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"/>
</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.

3.5.12 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"/>
</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.

3.5.13 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"/>
</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.

Book 4

Transactions

4.1 Transactional Messaging

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 *transactional controller* and *transactional resource* communicate over a *control link* which is established by the *transactional controller*. The `declare` and `discharge` messages are sent by the *transactional controller* over the *control link* to allocate and complete transactions respectively (they do not represent the demarcation of transactional work). No transactional work is allowed on the *control link*. Each transactional operation requested is explicitly identified with the desired transaction-id and therefore may occur on any link within the controlling Session, or, if permitted by the capabilities of the controller, any link on the controlling Connection. If the *control link* is closed while there exist non-discharged transactions it created, then all such transactions are immediately rolled back, and attempts to perform further transactional work on them will lead to failure.

4.2 Declaring a Transaction

The container acting as the transactional resource defines a special target that functions as a `coordinator`. The *transaction controller* establishes a control link to this target. Note that links to the `coordinator` cannot be resumed.

To begin transactional work, the transaction controller must obtain a transaction identifier from the resource. It does this by sending a message to the `coordinator` whose body consists of the `declare` type in a single `amqp-value` section. Other standard message sections such as the header section SHOULD be ignored. This message MUST NOT be sent settled as the sender is required to receive and interpret the outcome of the declare at the receiver. If the coordinator receives a `transfer` that has been settled by the sender, it should `detach` with an appropriate error.

If the declaration is successful, the coordinator responds with a disposition outcome of `declared` which carries the assigned identifier for the transaction.

If the coordinator was unable to perform the `declare` as specified by the transaction controller, the transaction coordinator MUST convey the error to the controller as a `transaction-error`. If the source for the link to the `coordinator` supports the rejected outcome, then the message MUST be rejected with this outcome carrying the `transaction-error`. If the source does not support the rejected outcome, the *transactional resource* MUST detach the link to the coordinator, with the detach performative carrying the `transaction-error`.

Transaction controllers SHOULD establish a control link that allows the rejected outcome.

```
Transaction Controller                  Transactional Resource
===============================================================================

ATTACH(name=txn-ctl,          --------->
       ...,
       target=
         Coordinator(
           capabilities=
             "amqp:local-transactions")
      )

                             <--------- ATTACH(name=txn-ctl,
                                               ...,
                                               target=
                                                 Coordinator(
                                                   capabilities=
                                                     ["amqp:local-transactions",
                                                      "amqp:multi-txns-per-ssn"]
                                                            )
                                              )

                             <--------- FLOW(...,handle=1, link-credit=1)

TRANSFER(delivery-id=0, ...) --------->
{ AmqpValue( Declare() ) }

                             <--------- DISPOSITION(first=0, last=0,
                                                    state=Declared(txn-id=0) )

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

Figure 4.1: Declaring a Transaction

4.3 Discharging a Transaction

The controller will conclude the transactional work by sending a `discharge` message (encoded in a single `amqp-value` section) to the coordinator. The controller indicates that it wishes to commit or rollback the transactional work by setting the *fail* flag on the `discharge` body. As with the `declare` message, it is an error if the sender sends the `transfer` pre-settled.

Should the coordinator be unable to complete the `discharge`, the coordinator MUST convey the error to the controller as a `transaction-error`. If the `source` for the link to the `coordinator` supports the `rejected` outcome, then the message MUST be `rejected` with this outcome carrying the `transaction-error`. If the `source` does not support the `rejected` outcome, the *transactional resource* MUST `detach` the link to the coordinator, with the `detach` performative carrying the `transaction-error`. Note that the coordinator MUST always be able to complete a `discharge` where the fail flag is set to true (since coordinator failure leads to rollback, which is what the controller is asking for).

```
Transaction Controller                  Transactional Resource
================================================================================

TRANSFER(delivery-id=0, ...) --------->
{ AmqpValue( Declare() ) }

                             <--------- DISPOSITION(first=0, last=0,
                                                    state=Declared(txn-id=0) )

                                  :
                          Transactional Work
                                  :


TRANSFER(delivery-id=57, ...) --------->
{ AmqpValue(
    Discharge(txn-id=0,
              fail=false)
           ) }

                             <--------- DISPOSITION(first=57, last=57,
                                                    state=Accepted() )

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

Figure 4.2: Discharging a Transaction

4.4 Transactional Work

Transactional work is described in terms of the message states defined in 3.3.1 Message States. Transactional work is formally defined to be composed of the following operations:

- *posting* a message at a target, i.e. making it *available*
- *acquiring* a message at a source, i.e. transitioning it to *acquired*
- *retiring* a message at a source, i.e. applying the terminal outcome

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 with the transactional delivery state `transactional-state` that combines a txn-id together with one of the terminal delivery states defined in section 3.4 Delivery State of the messaging specification. The `transactional-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.

4.4.1 Transactional Posting

If the transaction controller wishes to associate an outgoing transfer with a transaction, it must set the state of the transfer with a `transactional-state` carrying the appropriate transaction identifier. Note that if delivery is split across several transfer frames then all frames MUST be explicitly associated with the same transaction. It is an error for the controller to attempt to discharge a transaction against which a partial delivery has been posted. Should this happen, the control link MUST be terminated with the `transaction-rollback` error.

The effect of transactional posting is that the message does not become available at the destination node within the transactional resource until after the transaction has been (successfully) discharged.

```
Transaction Controller                  Transactional Resource
===============================================================================

TRANSFER(handle=0,           --------->
         delivery-id=0, ...)
{ AmqpValue( Declare() ) }

                             <--------- DISPOSITION(first=0, last=0,
                                                    state=Declared(txn-id=0) )


TRANSFER(handle=1,           --------->
         delivery-id=1,
         state=
           TransactionalState(
             txn-id=0) )
{ ... payload ... }

                             <--------- DISPOSITION(first=1, last=1,
                                                    state=TransactionalState(
                                                            txn-id=0,
                                                            outcome=Accepted())
                                                   )

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

Figure 4.3: Transactional Publish

On receiving a non-settled delivery associated with a live transaction, the transactional resource must inform the controller of the presumptive terminal outcome before it can successfully discharge the transaction. That is the resource must send a `disposition` performative which covers the posted transfer with the state of the delivery being a `transactional-state` with the correct transaction identified, and a terminal outcome. This informs the controller of the outcome that will be in effect at the point that the transaction is successfully discharged.

4.4.2 Transactional Retirement

The transaction controller may wish to associate the outcome of a delivery with a transaction. The delivery itself need not be associated with the same transaction as the outcome, or indeed with any transaction at all. However, the delivery MUST NOT be associated with a different *non discharged* transaction than the outcome. If this happens then the control link MUST be terminated with a `transaction-rollback` error.

To associate an outcome with a transaction the controller sends a `disposition` performative which sets the state of the delivery to a `transactional-state` with the desired transaction identifier and the outcome to be applied upon a successful discharge.

```
Transaction Controller                  Transactional Resource
===============================================================================

TRANSFER(handle=0,            --------->
         delivery-id=0, ...)
{ AmqpValue( Declare() ) }

                              <--------- DISPOSITION(first=0, last=0,
                                                     state=Declared(txn-id=0) )
FLOW(handle=2,                --------->
     link-credit=10)

                              <--------- TRANSFER(handle=2,
                                                  delivery-id=11,
                                                  state=null,
                                         { ... payload ... }

                                   :
                                   :

                              <--------- TRANSFER(handle=2,
                                                  delivery-id=20,
                                                  state=null,
                                         { ... payload ... }

DISPOSITION(first=11,         --------->
            last=20,
            state=TransactionalState(
                    txn-id=0,
                    outcome=Accepted())
           )
-------------------------------------------------------------------------------
```

Figure 4.4: Transactional Receive

On a successful `discharge`, the resource will apply the given outcome and may immediately settle the transfers. In the event of a controller initiated rollback (a `discharge` where the fail flag is set to true) or a resource initiated rollback (the `discharge` message being rejected, or the link to the `coordinator` being detached with an error), the outcome will not be applied, and the deliveries will still be "live" and will remain acquired by the controller - i.e. the resource should expect the controller to request a disposition for the delivery (either transactionally on a new transaction, or non-transactionally).

4.4.3 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 accommodate 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 *properties* map of the flow frame. When a transaction is discharged, the *txn-id* of any link endpoints will be cleared.

If the link endpoint does not support transactional acquisition, the link MUST be terminated with a `not-implemented` error.

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` indicates the failure of the transaction the resource MUST abort or complete the sending the remainder of the message before completing the discharge.

```
Transaction Controller                 Transactional Resource
===============================================================================

TRANSFER(handle=0,            --------->
         delivery-id=0, ...)
{ AmqpValue( Declare() ) }

                              <--------- DISPOSITION(first=0, last=0,
                                                     state=Declared(txn-id=0) )
FLOW(handle=2,                --------->
     link-credit=10,
     drain=true,
     properties={
       txn-id=0
     })

                              <--------- TRANSFER(handle=2,
                                                  delivery-id=11,
                                                  state=
                                                    TransactionalState(txn-id=0),
                                         { ... payload ... }

                                   :
                                   :

                              <--------- TRANSFER(handle=2,
                                                  delivery-id=20,
                                                  state=
                                                    TransactionalState(txn-id=0),
                                         { ... payload ... }


DISPOSITION(first=11,         --------->
            last=20,
            state=TransactionalState(
                    txn-id=0,
                    outcome=Accepted())
           )
-------------------------------------------------------------------------------
```

Figure 4.5: Transactional Acquisition

4.4.4 Interaction Of Settlement With Transactions

The Transport layer defines a notion of *settlement* which refers to the point at which the association of a delivery-tag to a delivery attempt is forgotten. Transactions do not in themselves change this notion, however the fact that transactional work may be rolled back does have implications for deliveries which the endpoint has marked as settled (and for which it therefore can no longer exchange state information using the previously allocated transport level identifiers).

4.4.4.1 Transactional Posting

Delivery Sent Settled By Controller

The delivered message will not be made available at the node until the transaction has been successfully discharged. If the transaction is rolled back then the delivery is not made available.

Should the resource be unable to process the delivery it MUST NOT allow the successful discharge of the associated transaction. This may be communicated by immediately destroying the controlling link on which the transaction was declared, or by rejecting any attempt to discharge the transaction where the fail flag is not set to true.

Delivery Sent Unsettled By Controller; Resource Settles

The resource MUST determine the outcome of the delivery before committing the transaction, and this MUST be communicated to the controller before the acceptance of a successful discharge. The outcome communicated by the resource MUST be associated with the same transaction with which the transfer from controller to resource was associated.

If the transaction is rolled back then the delivery is not made available at the target. If the resource can no longer apply the outcome that it originally indicated would be the result of a successful discharge then it MUST NOT allow the successful discharge of the associated transaction. This may be communicated by immediately destroying the controlling link on which the transaction was declared, or by rejecting any attempt to discharge the transaction where the fail flag is not set to true.

Delivery Sent Unsettled By Controller; Resource Does Not Settles

Behavior prior to discharge is the same as the previous case.

After a successful discharge, the state of unsettled deliveries at the resource MUST reflect the outcome that was applied. If the discharge was unsuccessful then no outcome should be associated with the unsettled deliveries. The controller SHOULD settle any outstanding unsettled deliveries in a timely fashion after the transaction has discharged.

4.4.4.2 Transactional Retirement

Here we consider the cases where the resource has sent messages to the controller in a non-transactional fashion. For the cases where the resource sends the messages transactionally, see **Transactional Acquisition** below.

Delivery Sent Unsettled By Resource; Controller Settles

Upon a successful discharge the outcome specified by the controller is applied at the source. Should the controller request a rollback or the discharge attempt be unsuccessful, then the outcome is not applied. At this point the controller can no longer influence the state of the delivery as it is settled, and the resource MUST apply the default outcome.

Delivery Sent Unsettled By Resource; Controller Does Not Settle

The resource may or may not settle the delivery before the transaction is discharged. If the resource settles the delivery before the discharge then the behavior after discharge is the same as the case above.

Upon a successful discharge the outcome is applied. Otherwise the state reverts to that which occurred before the controller sent its (transactional) disposition. The controller is free to update the state using subsequent transactional or non-transactional updates.

4.4.4.3 Transactional Acquisition

Delivery Sent Settled By Resource

In the event of a successful discharge the outcome applies at the resource, otherwise the acquisition and outcome are rolled back.

Delivery Sent Unsettled By Resource; Controller Sends Outcome

An outcome sent by the controller before the transaction has discharged MUST be associated with the same transaction. In the even of a successful discharge the outcome is applied at the source, otherwise both the acquisition and outcome are rolled back.

4.5 Coordination

4.5.1 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="capabilities" type="symbol" requires="txn-capability" multiple="true"/>
</type>
```

The coordinator type defines a special target used for establishing a link with a transaction coordinator.

Field Details

`capabilities` *the capabilities supported at the coordinator*

When sent by the transaction controller (the sending endpoint), indicates the desired capabilities of the coordinator. When sent by the resource (the receiving endpoint), defined the actual capabilities of the coordinator. Note that it is the responsibility of the transaction controller to verify that the capabilities of the controller meet its requirements. See `txn-capability`.

4.5.2 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="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 `declared` resulting outcome.

Field Details

`global-id` *global transaction id*

Specifies that the txn-id allocated by this declare MUST be associated work with the indicated global transaction. If not set, the allocated txn-id will associated work with a local transaction. This field MUST NOT be set if the Coordinator does not have the `distributed-transactions` capability. Note that the specification of distributed transactions within AMQP 1.0 will be provided separately in Book 6 Distributed Transactions.

4.5.3 Discharge

Message body for discharging a transaction.

```
<type name="discharge" class="composite" source="list">
    <descriptor name="amqp:discharge:list" code="0x00000000:0x00000032"/>
    <field name="txn-id" type="*" requires="txn-id" mandatory="true"/>
    <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

`txn-id` *identifies the transaction to be discharged*

`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. (Note that the specification for distributed transactions within AMQP 1.0 will be provided separately in Book 6 Distributed Transactions).

4.5.4 Transaction ID

```
<type name="transaction-id" class="restricted" source="binary" provides="txn-id"/>
```

A transaction-id may be up to 32 octets of binary data.

4.5.5 Declared

```
<type name="declared" class="composite" source="list" provides="delivery-state, outcome">
    <descriptor name="amqp:declared:list" code="0x00000000:0x00000033"/>
    <field name="txn-id" type="*" requires="txn-id" mandatory="true"/>
</type>
```

Indicates that a transaction identifier has successfully been allocated in response to a declare message sent to a transaction coordinator.

Field Details

`txn-id` *the allocated transaction id*

4.5.6 Transactional State

The state of a transactional message transfer.

```
<type name="transactional-state" class="composite" source="list" provides="delivery-state">
    <descriptor name="amqp:transactional-state:list" code="0x00000000:0x00000034"/>
    <field name="txn-id" type="*" mandatory="true" requires="txn-id"/>
    <field name="outcome" type="*" requires="outcome"/>
</type>
```

The transactional-state type defines a delivery-state that is used to associate a delivery with a transaction as well as to indicate which outcome is to be applied if the transaction commits.

Field Details

`txn-id` *identifies the transaction with which the state is associated*

`outcome` *provisional outcome*

This field indicates the provisional outcome to be applied if the transaction commits.

4.5.7 Transaction Capability

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

```
<type name="txn-capability" class="restricted" source="symbol" provides="txn-capability">
    <choice name="local-transactions" value="amqp:local-transactions"/>
    <choice name="distributed-transactions" value="amqp:distributed-transactions"/>
    <choice name="promotable-transactions" 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"/>
</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.

4.5.8 Transaction Error

Symbols used to indicate transaction errors.

```
<type name="transaction-error" class="restricted" source="symbol" provides="error-condition">
    <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 5

Security

5.1 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 section 5.3 SASL 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.

5.2 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 1 (TLS-MAJOR), 0 (TLS-MINOR), 0 (TLS-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.
- Implementations MAY choose to use TLS with unidirectional shutdown, i.e. an application initiating shutdown using close_notify is not obliged to wait for the peer to respond, and MAY close the write-half of the TCP socket.

5.2.1 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).

5.2.2 Constant Definitions

TLS-MAJOR	1	major protocol version.
TLS-MINOR	0	minor protocol version.
TLS-REVISION	0	protocol revision.

5.3 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 1 (SASL-MAJOR), 0 (SASL-MINOR), 0 (SASL-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
```

5.3.1 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 5.1: 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="sasl-frame" . Receipt of an empty frame is an irrecoverable error.

5.3.2 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 5.2: 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.

5.3.3 Security Frame Bodies

5.3.3.1 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="sasl-server-mechanisms" type="symbol" multiple="true" mandatory="true"/>
</type>
```

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

Field Details

`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.

5.3.3.2 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="mechanism" type="symbol" 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

`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.

5.3.3.3 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="challenge" type="binary" mandatory="true"/>
</type>
```

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

Field Details

`challenge` *security challenge data*

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

5.3.3.4 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="response" type="binary" mandatory="true"/>
</type>
```

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

Field Details

`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.

5.3.3.5 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="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

`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.

5.3.3.6 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.

5.3.4 Constant Definitions

SASL-MAJOR	1	major protocol version.
SASL-MINOR	0	minor protocol version.
SASL-REVISION	0	protocol revision.