

AMQP 1.0 DRAFT

for Review Only

Revision: Recommendation (*draft*)

Creation Date: 2010-05-19 08:15:01
Subversion Rev: 1090

UNPUBLISHED CONFIDENTIAL MATERIAL OF
THE AMQP WORKING GROUP
DISTRIBUTION TO MEMBERS AND REVIEWERS ONLY

License

This document contains a DRAFT of the AMQP1-0 specification and is for review only. It does not constitute a published AMQP specification.

The contents of this document are licensed under the terms of use of the amqp.org website given here:

<http://jira.amqp.org/confluence/display/AMQP/Terms+of+Use>

Table of Contents

1 Introduction to AMQP.....	7
1.1 Overview.....	7
1.2 Rationale and Use Cases.....	7
1.3 How to Read the Standard.....	8
Book I - AMQP Layering.....	10
1 Layering Diagram.....	11
Book II - Data Types.....	12
1 Type System.....	13
1.1 Primitive Types.....	13
1.2 Decoding Primitive Types.....	13
1.3 Described Types.....	14
1.4 Descriptor Values.....	14
2 Type Encodings.....	15
2.1 Fixed Width.....	17
2.2 Variable Width.....	17
2.3 Compound.....	18
2.4 Array.....	18
2.5 List of Encodings.....	19
3 Composite Types.....	21
3.1 List Encoding.....	21
3.2 Map Encoding.....	22
Book III - Transport.....	23
1 Transport.....	24
2 Version Negotiation.....	27
3 Framing.....	29
3.1 Frame Layout.....	29
3.2 AMQP Frames.....	30
4 Connections.....	31
4.1 Opening a Connection.....	31
4.2 Pipelined Open.....	32
4.3 Closing a Connection.....	32
4.4 Simultaneous Close.....	33
4.5 Connection States.....	33
4.6 Connection State Diagram.....	34
5 Sessions.....	36
5.1 Establishing a Session.....	36
5.2 Ending a Session.....	37
5.3 Simultaneous End.....	37
5.4 Session Errors.....	38
5.5 Session States.....	38

6 Links.....	40
6.1 Naming a Link.....	40
6.2 Link Handles.....	40
6.3 Establishing a Link.....	41
6.4 Resuming a Link.....	42
6.5 Closing a Link.....	43
6.6 Flow Control.....	43
6.7 Synchronous Get.....	46
6.8 Asynchronous Notification.....	47
6.9 Stopping a Link.....	48
6.10 Transferring a Message.....	48
6.11 Resuming Transfers.....	53
6.12 Message Fragmentation.....	53
7 Frame-bodies.....	55
7.1 open (negotiate Connection parameters).....	55
7.2 begin (begin a Session on a channel).....	56
7.3 attach (attach a Link to a Session).....	57
7.4 flow (update link state).....	58
7.5 transfer (transfer a Message).....	59
7.6 disposition (inform remote peer of transfer state changes).....	59
7.7 detach (detach the Link Endpoint from the Session).....	59
7.8 end (end the Session).....	60
7.9 close (signal a Connection close).....	60
8 Definitions.....	61
8.1 role: boolean(link endpoint role).....	61
8.2 handle: uint(the handle of a Link).....	61
8.3 linkage (the source and target for a link).....	61
8.4 link-expiry-policy: symbol(expiry policy for a link endpoint).....	61
8.5 seconds: uint(a duration measured in seconds).....	62
8.6 flow-state	62
8.7 delivery-tag: binary.....	62
8.8 transfer-number: sequence-no.....	62
8.9 sequence-no: uint(32-bit RFC-1982 serial number).....	62
8.10 extent	63
8.11 fragment (a Message fragment).....	63
8.12 fields: map(a mapping from field name to value).....	63
8.13 options: fields(options map).....	63
8.14 error-mode: symbol(behaviors for link errors).....	64
8.15 error (details of an error).....	64
8.16 amqp-error: symbol(shared error conditions).....	65
8.17 connection-error: symbol(symbols used to indicate connection error conditions).....	65
8.18 session-error: symbol(symbols used to indicate session error conditions).....	65
8.19 link-error: symbol(symbols used to indicate link error conditions).....	66
Book IV - Messaging.....	68
1 Introduction.....	69

2 Message Format.....	70
2.1 section-codes: uint.....	71
2.2 header (transport headers for a Message).....	71
2.3 properties (immutable properties of the Message).....	73
2.4 footer (transport footers for a Message).....	74
2.5 message-attributes: map(message annotations).....	74
3 Transfer State.....	75
3.1 transfer-state (the state of a message transfer).....	75
3.2 accepted (the accepted outcome).....	75
3.3 rejected (the rejected outcome).....	75
3.4 released (the released outcome).....	76
3.5 modified (the modified outcome).....	76
4 Distribution Nodes.....	77
4.1 Message States.....	77
5 Sources and Targets.....	78
5.1 Filtering Messages.....	78
5.2 Distribution Modes.....	78
5.3 source	78
5.4 target	80
5.5 std-dist-mode: symbol(Link distribution policy).....	80
5.6 filter (the predicate to filter the Messages admitted onto the Link).....	80
5.7 filter-set: map.....	81
5.8 delete-on-close (lifetime of dynamic Node scoped to lifetime of link which caused creation).....	81
5.9 delete-on-no-links (lifetime of dynamic Node scoped to existence of links to the Node)..	81
5.10 delete-on-no-messages (lifetime of dynamic Node scoped to existence of messages on the Node).....	81
5.11 delete-on-no-links-or-messages (lifetime of Node scoped to existence of messages on or links to the Node).....	82
Book V - Transactions.....	83
1 Transactions.....	84
1.1 Transactional Acquisition.....	85
2 Coordination.....	86
2.1 coordinator (target for communicating with a transaction coordinator).....	86
2.2 declare (message body for declaring a transaction id).....	86
2.3 discharge (message body for discharging a transaction).....	86
2.4 txn-capabilities: symbol(symbols indicating (desired/available) capabilities of a transaction coordinator).....	87
2.5 transaction-errors: symbol(symbols used to indicate transaction errors).....	87
Book VI - Security.....	88
1 Security Layers.....	89
2 TLS.....	90
3 SASL.....	91
3.1 SASL Negotiation.....	91
3.2 Security Frame Bodies.....	92

3.3 sasl-mechanisms (advertise available sasl mechanisms).....	92
3.4 sasl-init (initiate sasl exchange).....	92
3.5 sasl-challenge (security mechanism challenge).....	93
3.6 sasl-response (security mechanism response).....	93
3.7 sasl-outcome (indicates the outcome of the sasl dialog).....	93
3.8 sasl-code: ubyte(codes to indicate the outcome of the sasl dialog).....	94

1 Introduction to AMQP

1.1 Overview

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

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

1.2 Rationale and Use Cases

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

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

<i>AMQP Release</i>	<i>User Requirement</i>
	<i>Ubiquity</i>
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
	<i>Safety</i>
1.0	Infrastructure for a secure and trusted global transaction network <ul style="list-style-type: none">• 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
	<i>Fidelity</i>
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
	<i>Applicability</i>
	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 $y \geq x$
- Future Stable extended (broker-broker) wire protocol so that brokers do not require upgrade during 1.x feature evolution: Any two brokers versions 1.x, 1.y can communicate using protocol 1.x if $x < y$
- 1.0 Layered architecture, so features & network transports can be independently extended by separated communities of use, enabling business integration with other systems without coordination with the AMQP Working Group

Manageability

- 1.0 Binary WIRE protocol so that it can be ubiquitous, fast, embedded (XML can be layered on top), enabling management to be provided by encapsulating systems (e.g. O/S, middleware, phone)
- 1.0 Scalable, so that it can be a basis for high performance fault-tolerant lossless messaging infrastructure, i.e. without requiring other messaging technology
- Future Interaction with the message delivery system is possible, sufficient to integrate with prevailing business operations that administer messaging systems using management standards.
- Future Intermediated: supports routing and relay management, traffic flow management and quality of service management
- Future Decentralized deployment with independent local governance
- Future Global addressing standardizing end to end delivery across any network scope

1.3 How to Read the Standard

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

Book I Overview of the AMQP Layering

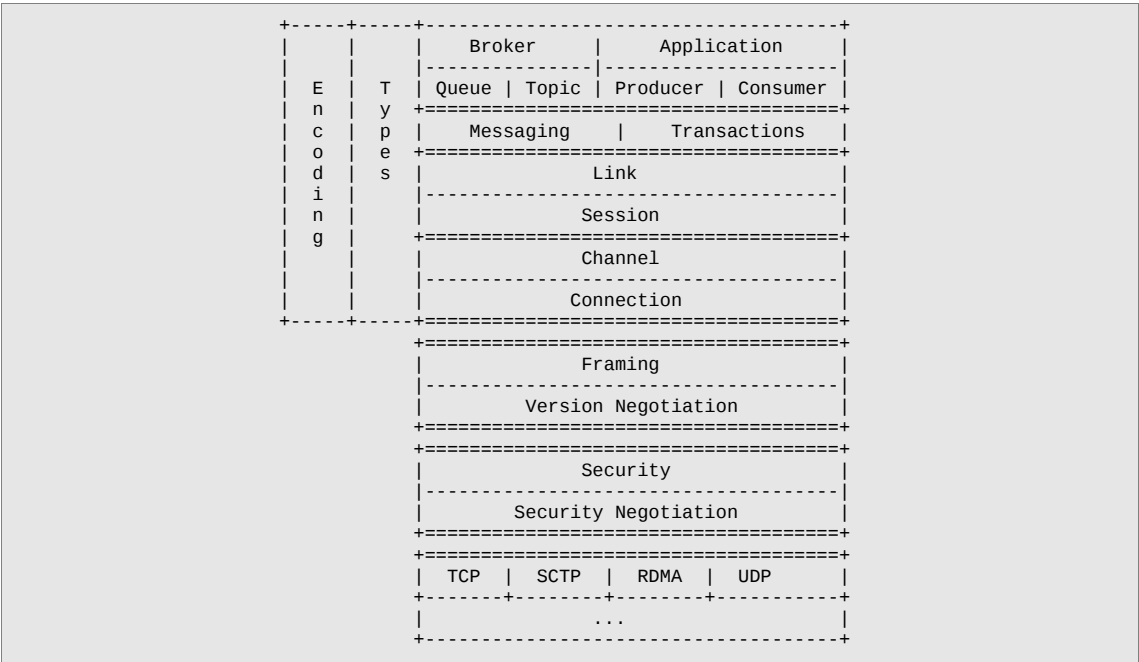
Book II Defines the AMQP Type System

Book III Defines the AMQP Transport Layer
Book IV Defines the AMQP Messaging Layer
Book V Defines the AMQP Transaction Layer
Book VI Defines the AMQP Security Layers

Book I - AMQP Layering

1 Layering Diagram

TODO: ADD TEXT DESCRIBING LAYERING



Book II - Data Types

1 Type System

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

1.1 Primitive Types

The following primitive types are defined:

null:	Indicates an empty value.
boolean:	Represents a true or false value.
ubyte:	Integer in the range 0 to $2^8 - 1$.
ushort:	Integer in the range 0 to $2^{16} - 1$.
uint:	Integer in the range 0 to $2^{32} - 1$.
ulong:	Integer in the range 0 to $2^{64} - 1$.
byte:	Integer in the range $-(2^7)$ to $2^7 - 1$.
short:	Integer in the range $-(2^{15})$ to $2^{15} - 1$.
int:	Integer in the range $-(2^{31})$ to $2^{31} - 1$.
long:	Integer in the range $-(2^{63})$ to $2^{63} - 1$.
float:	32-bit floating point number (IEEE 754-2008 binary32).
double:	64-bit floating point number (IEEE 754-2008 binary64).
decimal32:	32-bit decimal number (IEEE 754-2008 decimal32).
decimal64:	64-bit decimal number (IEEE 754-2008 decimal64).
char:	A single unicode character.
timestamp:	An absolute point in time.
uuid:	A universally unique id as defined by RFC-4122 section 4.1.2.
binary:	A sequence of octets.
string:	A sequence of unicode characters.
symbol:	Symbols are values from a constrained domain. Although the set of possible domains is open-ended, typically the both number and size of symbols in use for any given application will be small, e.g. small enough that it is reasonable to cache all the distinct values.
list:	A sequence of polymorphic values.
map:	A polymorphic mapping from distinct keys to values.

1.2 Decoding Primitive Types

For any given programming language there may not be a direct equivalence between the available native language types and the AMQP types. The list of mappings between AMQP

types and programming language types can be found here: <http://www.amqp.org/spec/1-0-draft/type-mappings/mapped-languages.xml>

TODO: GENERATE THE URL BASED OF THE PROTOCOL REVISION

1.3 Described Types

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

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

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

1.4 Descriptor Values

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

The namespace for both symbolic and numeric descriptors is divided into distinct domains. Each domain has a defined symbol and/or 4 byte numeric id assigned by the AMQP working group. Descriptors are then assigned within each domain according to the following rules:

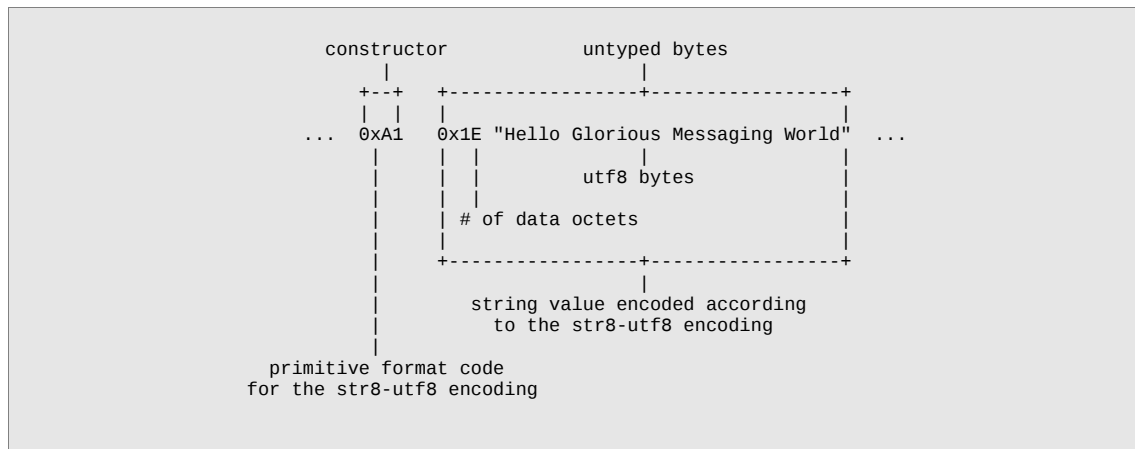
symbolic `<domain>:<name>`
descriptors:

numeric `(domain-id << 32) | descriptor-id`
descriptors:

2 Type Encodings

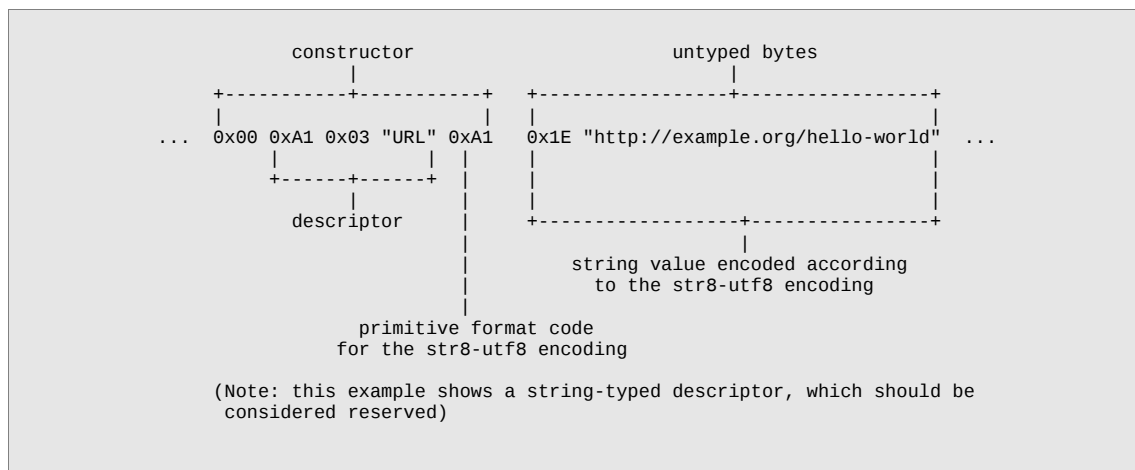
An AMQP encoded data stream consists of untyped bytes with embedded constructors. The embedded constructor indicates how to interpret the untyped bytes that follow. Constructors can be thought of as functions that consume untyped bytes from an open ended byte stream and construct a typed value. An AMQP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

Primitive Format Code (String)



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

Described Format Code (URL)



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

Constructor BNF

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

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

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

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

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

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

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

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

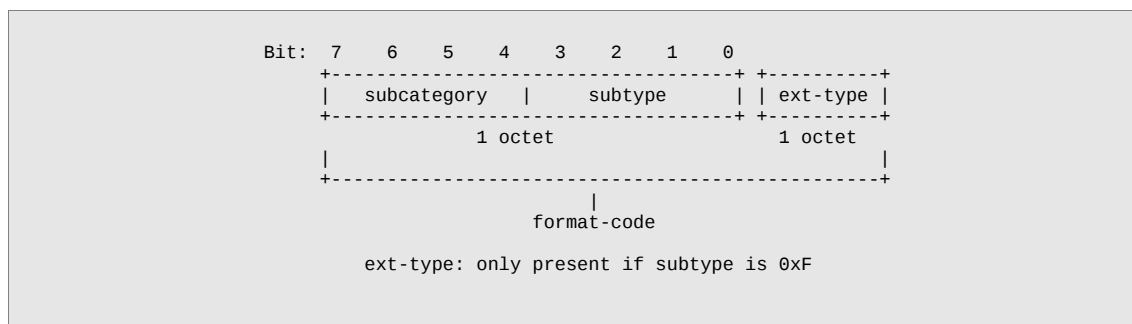
Fixed Width: The size of fixed-width data is determined based solely on the subcategory of the format code for the fixed width value.

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

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

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

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



The following table describes the subcategories of format-codes:

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

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

2.1 Fixed Width

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

n OCTETS	
+-----+	
data	
+-----+	
Subcategory	n
0x4	0
0x5	1
0x6	2
0x7	4
0x8	8
0x9	16

2.2 Variable Width

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

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

n OCTETs size OCTETs	
-----+-----	
size value	
-----+-----	
Subcategory	n
=====	
0xA	1
0xB	4

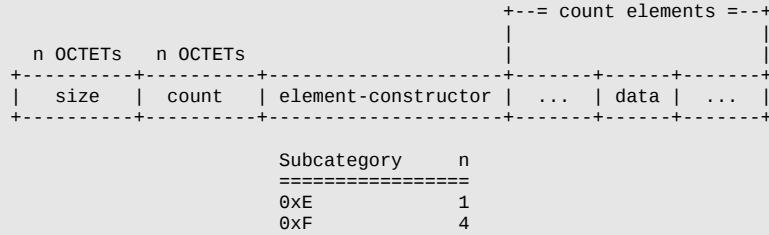
2.3 Compound

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

n OCTETs n OCTETs		-----= count items -----	
-----+-----			
size count		... / item \ ...	
-----+-----		-----+-----	
		constructor data	
		-----+-----	
Subcategory	n		
=====			
0xC	1		
0xD	4		

2.4 Array


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



2.5 List of Encodings

Type	Encoding	Code	Category/Width	Description
null		0x40	fixed/0	The null value.
boolean	true	0x41	fixed/0	The boolean value true.
boolean	false	0x42	fixed/0	The boolean value false.
ubyte		0x50	fixed/1	8-bit unsigned integer.
ushort		0x60	fixed/2	16-bit unsigned integer in network byte order.
uint		0x70	fixed/4	32-bit unsigned integer in network byte order.
ulong		0x80	fixed/8	64-bit unsigned integer in network byte order.
byte		0x51	fixed/1	8-bit two's-complement integer.
short		0x61	fixed/2	16-bit two's-complement integer in network byte order.
int		0x71	fixed/4	32-bit two's-complement integer in network byte order.
long		0x81	fixed/8	64-bit two's-complement integer in network byte order.
float	ieee-754	0x72	fixed/4	IEEE 754-2008 binary32.
double	ieee-754	0x82	fixed/8	IEEE 754-2008 binary64.
decimal32	ieee-754	0x74	fixed/4	IEEE 754-2008 decimal32.
decimal64	ieee-754	0x84	fixed/8	IEEE 754-2008 decimal64.
char	utf32	0x73	fixed/4	A UTF-32 encoded unicode character.
timestamp	ms64	0x83	fixed/8	Encodes a point in time using a 64 bit signed integer representing milliseconds since Midnight Jan 1, 1970 UTC. For the purpose of this representation, milliseconds are taken to be $(1/(24*60*60*1000))$ th of a day.
uuid		0x98	fixed/16	UUID as defined in section 4.1.2 of RFC-4122.
binary	vbin8	0xa0	variable/1	Up to $2^8 - 1$ octets of binary data.
binary	vbin32	0xb0	variable/4	Up to $2^{32} - 1$ octets of binary data.
string	str8-utf8	0xa1	variable/1	Up to $2^8 - 1$ octets worth of UTF-8 unicode.
string	str8-utf16	0xa2	variable/1	Up to $2^8 - 1$ octets worth of UTF-16 unicode.
string	str32-utf8	0xb1	variable/4	Up to $2^{32} - 1$ octets worth of UTF-8 unicode.
string	str32-utf16	0xb2	variable/4	Up to $2^{32} - 1$ octets worth of UTF-16 unicode.
symbol	sym8	0xa3	variable/1	Up to $2^8 - 1$ seven bit ASCII characters representing a symbolic value.
symbol	sym32	0xb3	variable/4	Up to $2^{32} - 1$ seven bit ASCII characters representing a symbolic value.
list	list8	0xc0	compound/1	Up to $2^8 - 1$ list elements with total size less than 2^8 octets.
list	list32	0xd0	compound/4	Up to $2^{32} - 1$ list elements with total size less than 2^{32} octets.
list	array8	0xe0	array/1	Up to $2^8 - 1$ array elements with total size less than 2^8 octets.
list	array32	0xf0	array/4	Up to $2^{32} - 1$ array elements with total size less than 2^{32} octets.
map	map8	0xc1	compound/1	Up to $2^8 - 1$ octets of encoded map data. 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	map32	0xd1	compound/4	 <p>Up to $2^{32} - 1$ octets of encoded map data. See map8 above for a definition of encoded map data.</p>
-----	-------	------	------------	--

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:

Example Composite Type

```
<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>
```

The *mandatory* attribute of a field description controls whether a null element value is permitted in the representation. The *multiple* attribute of a field description controls whether multiple element values are permitted in the representation. A single element of the type specified in the field description is always permitted, as is a null element value. Multiple values are represented by the use of a described list (of any valid encoding). The descriptor for the list **MUST** be the boolean value *true*, and every element in the list **MUST** be of the type matching the field description.

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

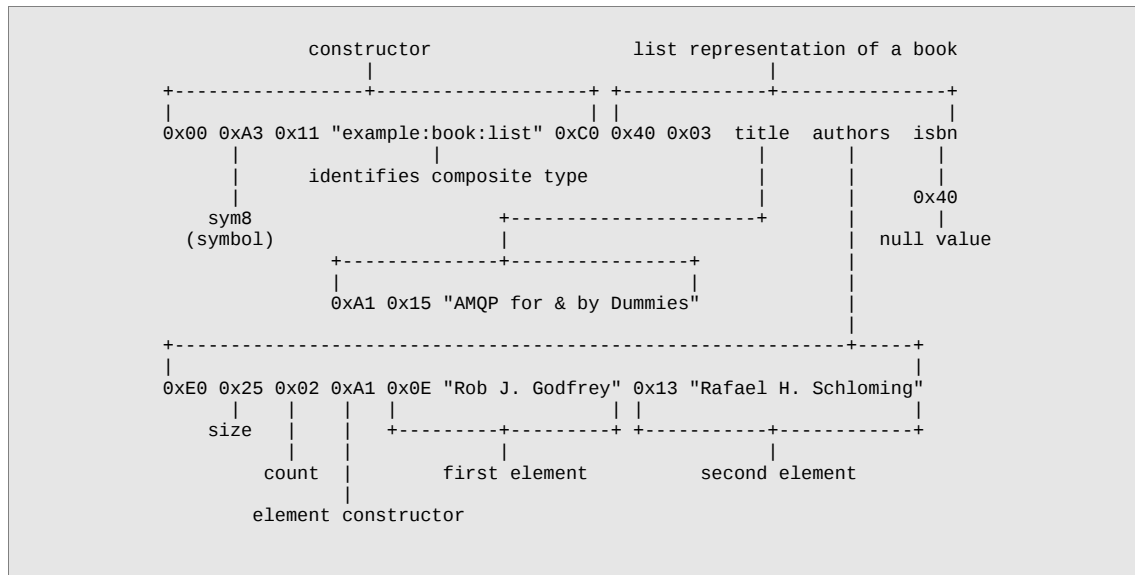
3.1 List Encoding

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

The described list shown below is an example composite value of the *book* type defined above. A trailing null element corresponding to the absence of an isbn value is depicted in the example,

but may optionally be omitted according to the encoding rules.

Example Composite Value



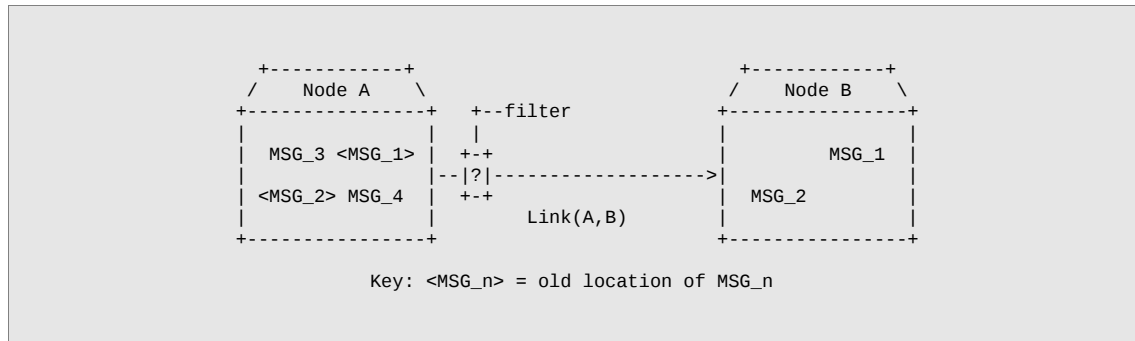
3.2 Map Encoding

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

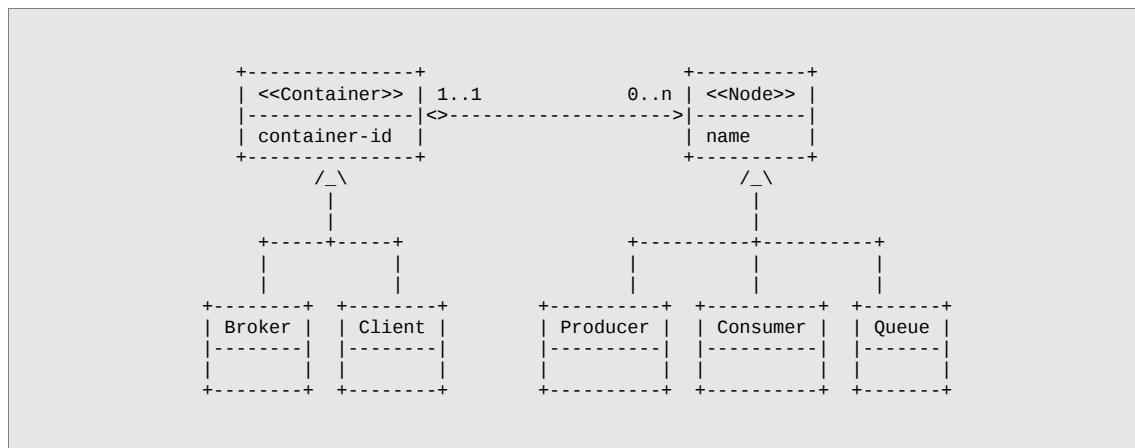
Book III - Transport

1 Transport

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



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.

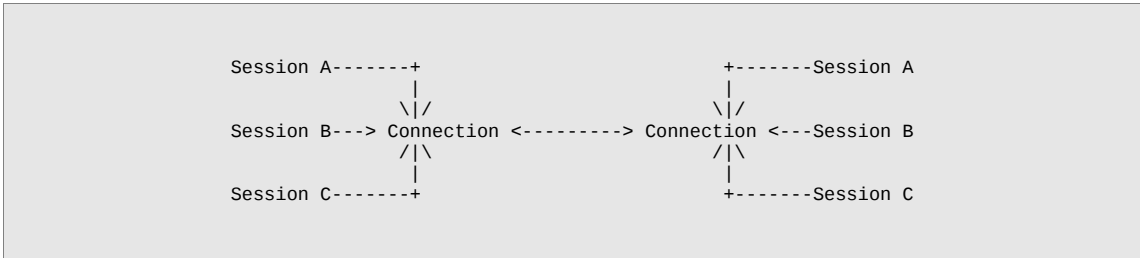


The AMQP Transport Specification defines a peer-to-peer protocol for transferring Messages between Nodes in the AMQP network. This portion of the specification is not concerned with the internal workings of any sort of Node, and only deals with the mechanics of unambiguously transferring a Message from one Node to another.

Containers communicate via *Connections*. An AMQP Connection consists of a full-duplex, reliably ordered sequence of *Frames*. The precise requirement for a Connection is that if the n-th Frame arrives, all Frames prior to n **MUST** also have arrived. It is assumed Connections are transient and may fail for a variety of reasons resulting in the loss of an unknown number of

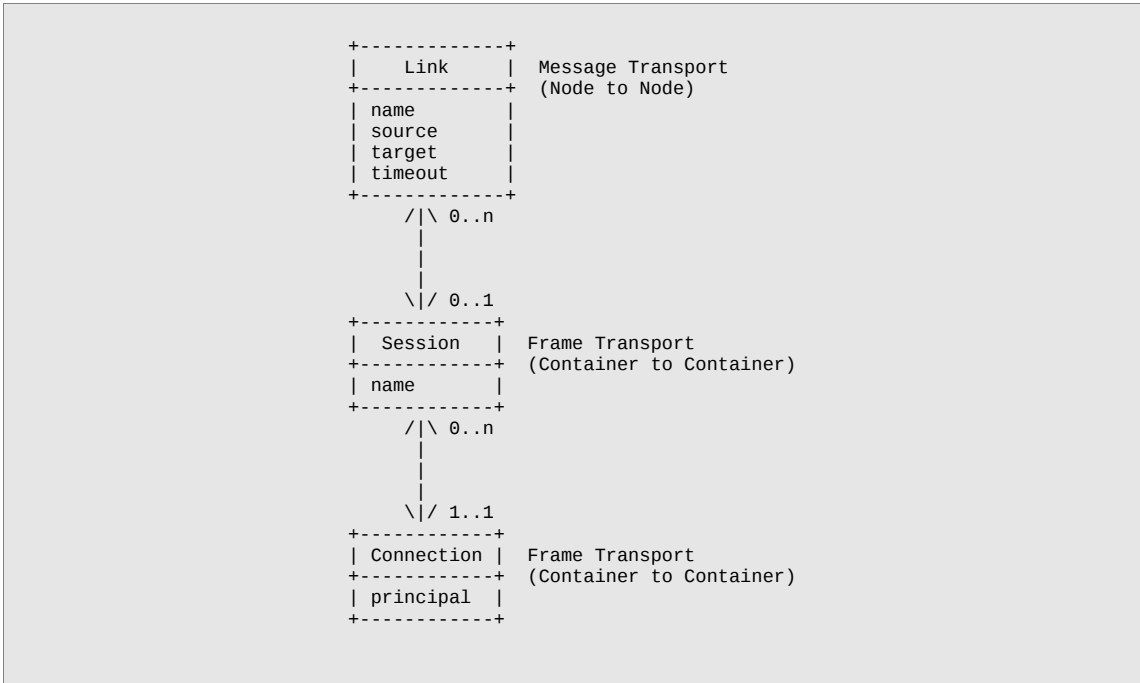
frames, but they are still subject to the aforementioned ordered reliability criteria. This is similar to the guarantee that TCP or SCTP provides for byte streams, and the specification defines a framing system used to parse a byte stream into a sequence of Frames for use in establishing an AMQP Connection over TCP or SCTP.

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



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

TODO: MAKE THIS DIAGRAM LESS USELESS



A *Frame* is the unit of work carried on the wire. Connections have a negotiated maximum frame size allowing byte streams to be easily defragmented into complete frame bodies representing the

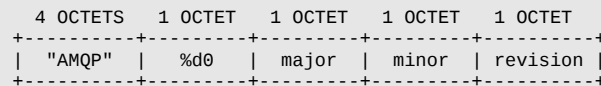
independently parsable units formally defined in the frame-bodies section. The following table lists all frame bodies and defines which endpoints handle them.

TODO: MAKE THESE XREFS SOMEHOW

Frame	Connection	Session	Link
=====	=====	=====	=====
open	H		
begin	I	H	
attach		I	H
flow		I	H
transfer		I	H
disposition		I	H
detach		I	H
end	I	H	
close	H		
-----	-----	-----	-----
Key:			
H: handled by the endpoint			
I: intercepted (endpoint examines the frame, but delegates further processing to another endpoint)			

2 Version Negotiation

Prior to sending any frames on a Connection, each peer **MUST** start by sending a protocol header that indicates the protocol version used on the Connection. The protocol header consists of the upper case ASCII letters "AMQP" followed by a protocol id of zero, followed by three unsigned bytes representing the major, minor, and revision of the protocol version (currently 1, 0, 0). In total this is an 8-octet sequence:



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

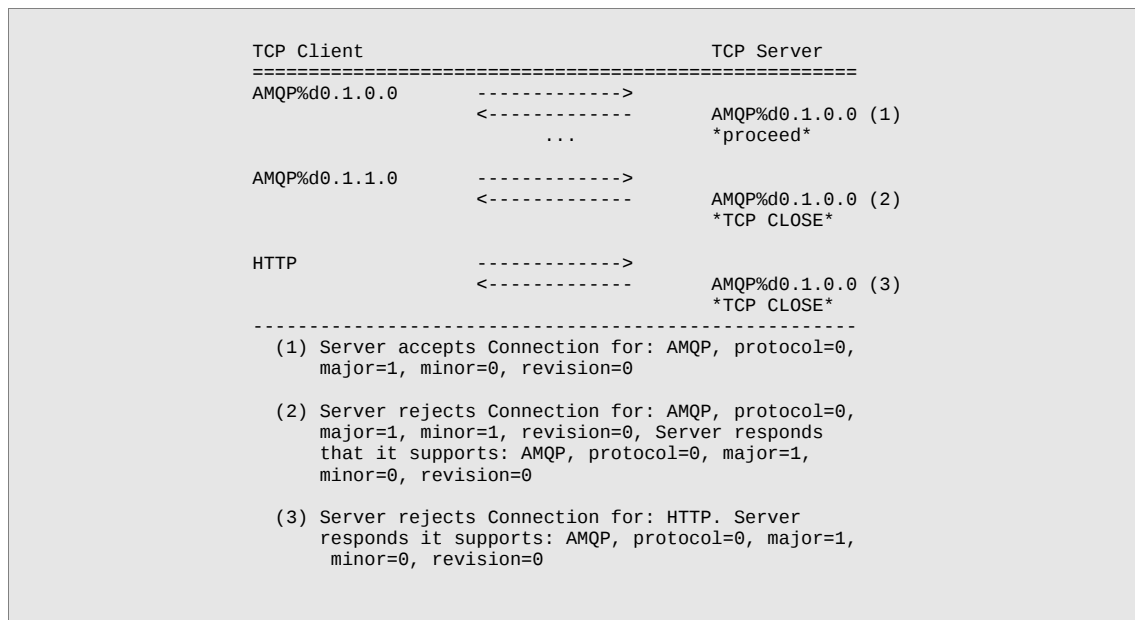
The AMQP peer which acted in the role of the TCP client (i.e. the peer that opened the Connection) **MUST** immediately send its outgoing protocol header on establishment of the TCP Session. The AMQP peer which acted in the role of the TCP server **MAY** elect to wait until receiving the incoming protocol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

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

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

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

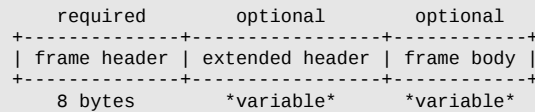
Version Negotiation Examples



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

3 Framing

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



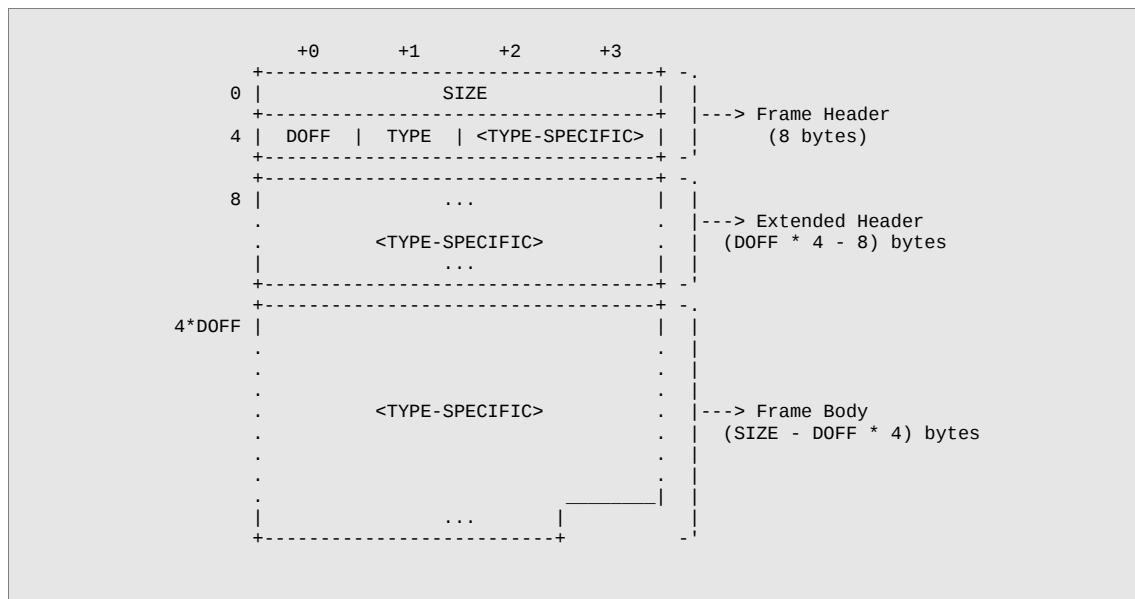
frame header: The frame header is a fixed size (8 byte) structure that precedes each frame. The frame header includes mandatory information required to parse the rest of the frame including size and type information.

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

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

3.1 Frame Layout

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



SIZE: Bytes 0-3 of the frame header contain the frame size. This is an unsigned 32-bit integer that **MUST** contain the total frame size of the frame header, extended header, and frame body. The frame is malformed if the size is less than the the size of the required frame header (8 bytes).

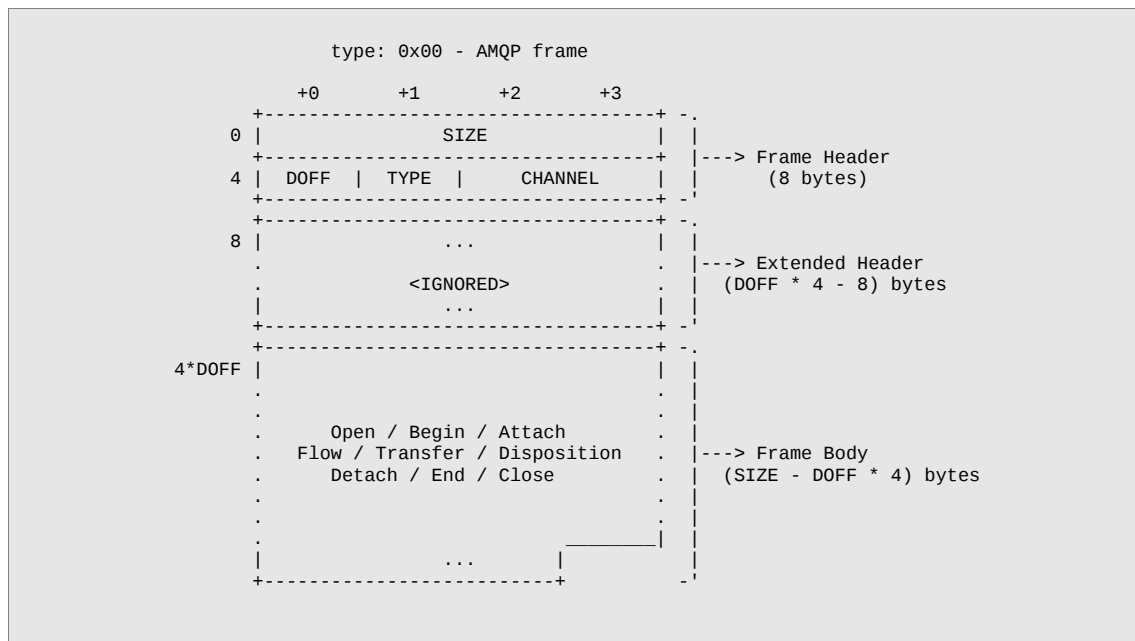
DOFF: Byte 4 of the frame header is the data offset. This gives the position of the

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

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

3.2 AMQP Frames

The AMQP frame type defines header bytes 6 and 7 to contain a channel number. **TODO: NEED TO REFERENCE WHAT A CHANNEL IS** The AMQP frame type defines bodies encoded as described types in the AMQP type system.



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

4 Connections

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

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

```

                                     +-----OCHE X: 1
                                     |
                                     +-----OCHE Y: 7
                                     |
<=== Frame[CH=1], Frame[CH=7] <====+
===> Frame[CH=0], Frame[CH=1] ===>+
                                     |
                                     +----->0: ICHE A
                                     |
                                     +----->1: ICHE B

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

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

```

                                     +-----BIDI/O: 7
                                     |
<=== Frame[CH=1], Frame[CH=7] <====+
===> Frame[CH=0], Frame[CH=1] ===>+
                                     |
                                     +----->1: BIDI/I

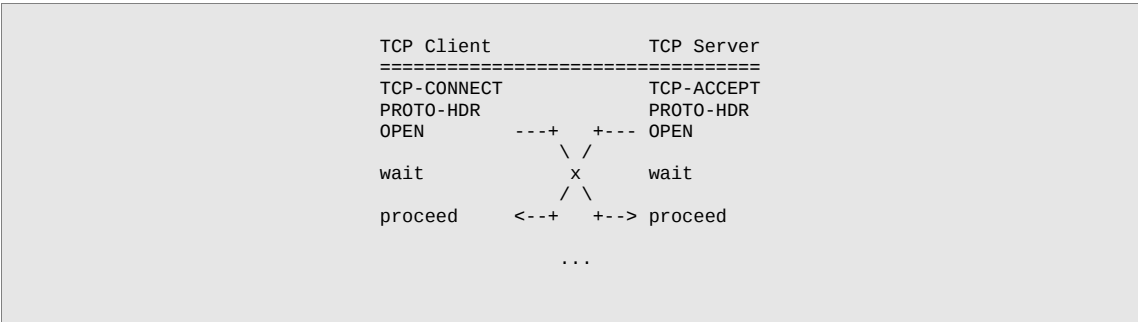
BIDI/I: Incoming half of a single bidirectional endpoint
BIDI/O: Outgoing half of a single bidirectional endpoint
```

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.

4.1 Opening a Connection

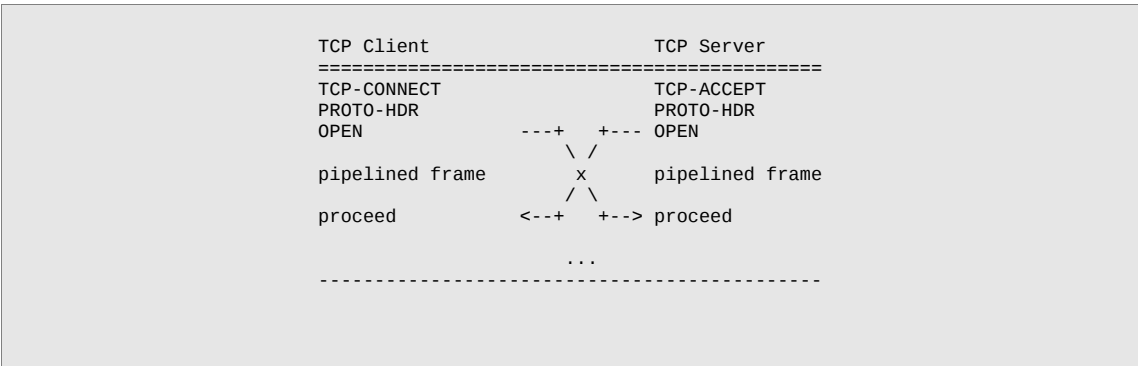
Each AMQP Connection begins with an exchange of capabilities and limitations, including the maximum frame size. Prior to any explicit negotiation, the maximum frame size is 4096. After

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



4.2 Pipelined Open

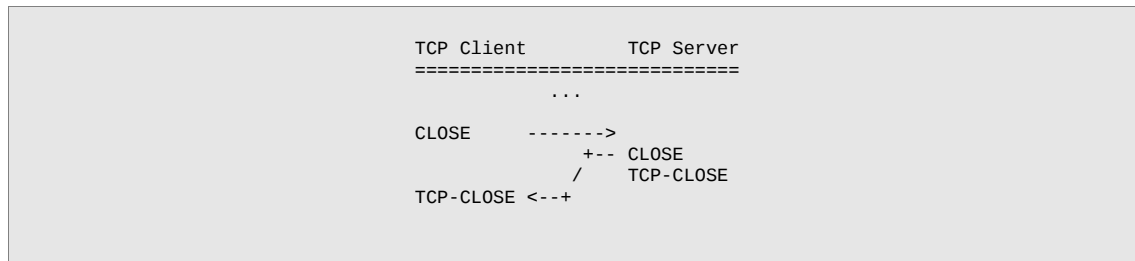
For applications that use many short-lived Connections, it may be desirable to pipeline the Connection negotiation process. A peer may do this by starting to send subsequent frames before receiving the partner's Connection header or open frame. This is permitted so long as the pipelined frames are known 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.



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

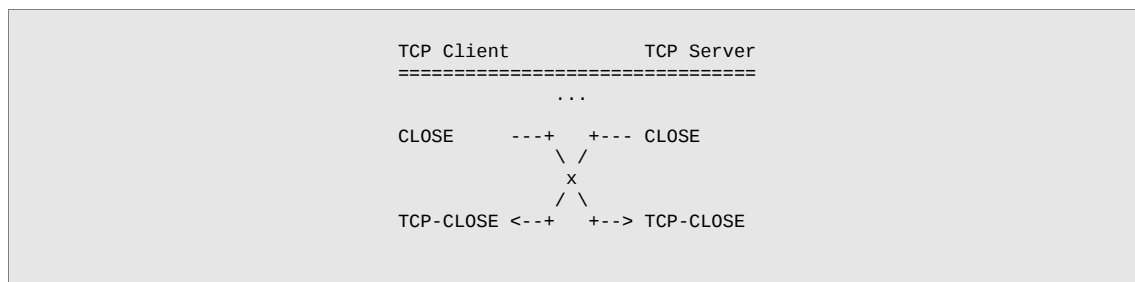
4.3 Closing a Connection

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



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.



4.5 Connection States

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

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

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

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

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

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

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

CLOSE_PIPE: In this state we have send and received the Connection header, sent an

open frame, any pipelined Connection traffic, and the `close` frame, but we have not yet received an `open` frame.

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

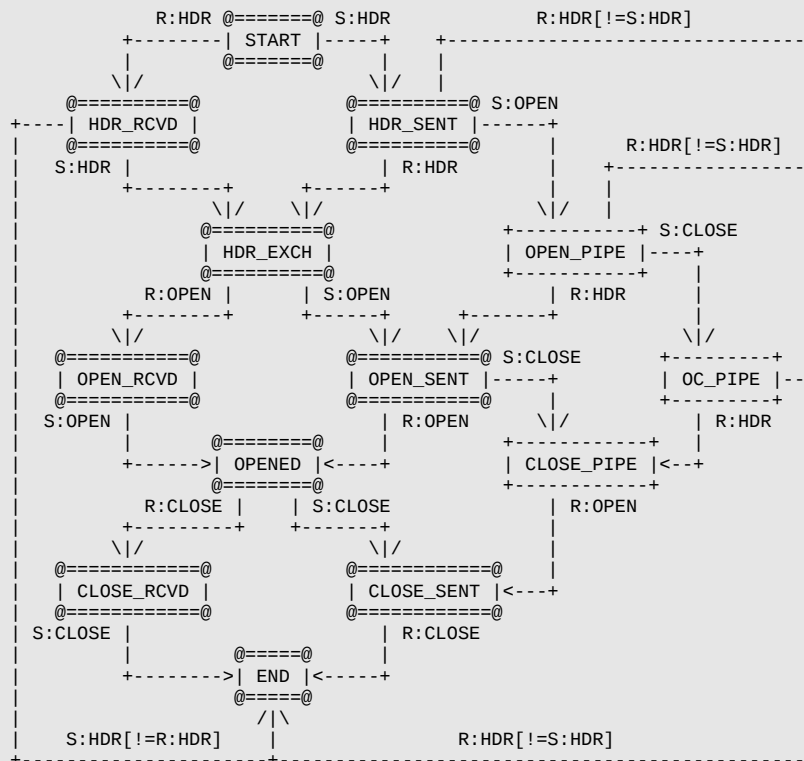
CLOSE_RCVD: In this state we have received a `close` frame indicating that our partner has initiated a close. This means we will never have to read anything more from this Connection, however we can continue to write frames onto the Connection. If desired, an implementation could do a TCP half-close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

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

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

4.6 Connection State Diagram

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



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

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

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

5 Sessions

A Session is a bidirectional sequential conversation between two containers that provides a grouping for related links. Sessions serve as the context for active 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.



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

TODO: PICTURE

Sessions additionally provide transport level flow control for messages transferred along attached links. This bounds the amount of unsettled transfer state to the amount required to accomodate the bandwidth delay product of the underlying communication fabric.

TODO: PICTURE

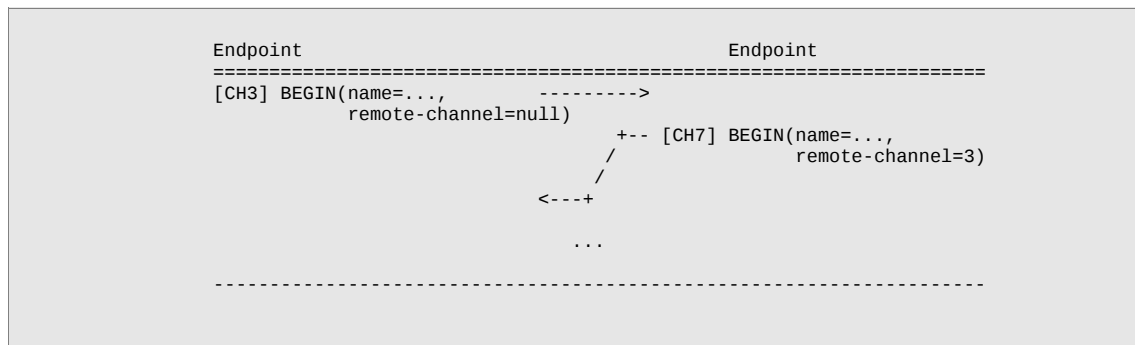
A session provides its bidirectional communication by mapping together two channels, each one forming a half-session.

TODO: PICTURE

5.1 Establishing a Session

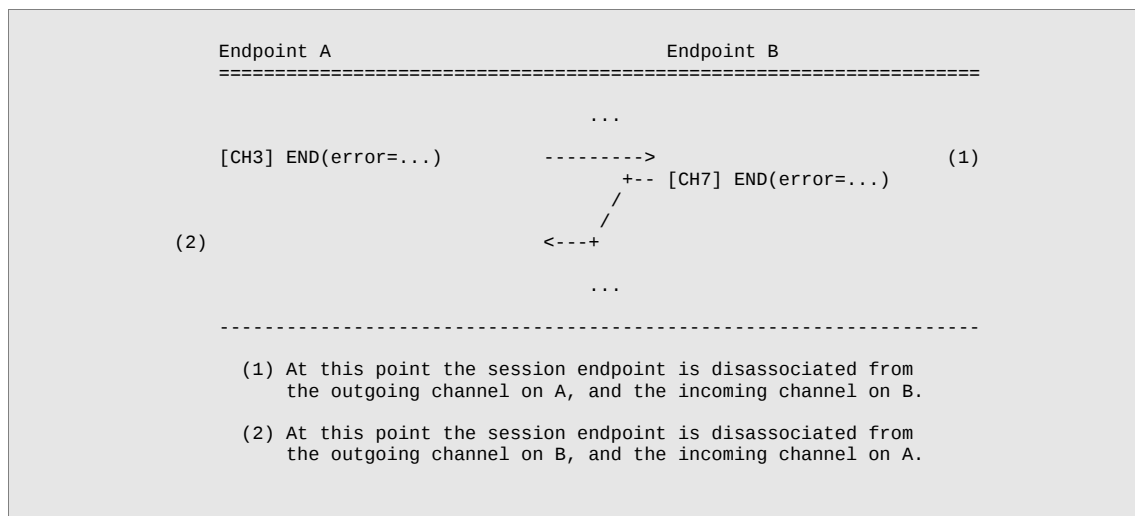
Sessions are established by creating a Session Endpoint, assigning it to an unused channel number, and sending a begin announcing the association of the Session Endpoint with the outgoing channel. Upon receiving the begin the partner will check the remote-channel field and find it empty. This indicates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

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



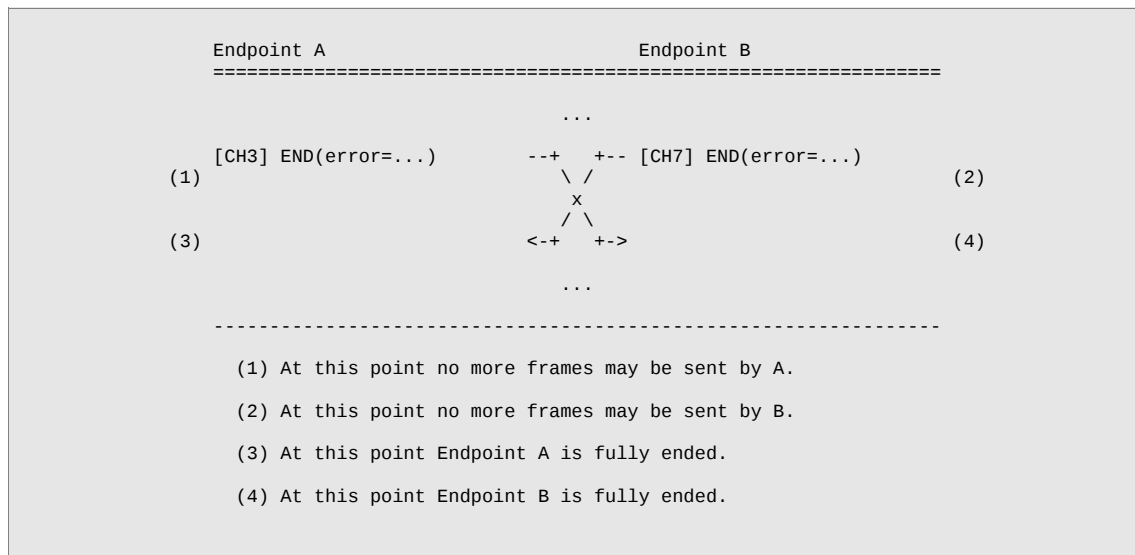
5.2 Ending a Session

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



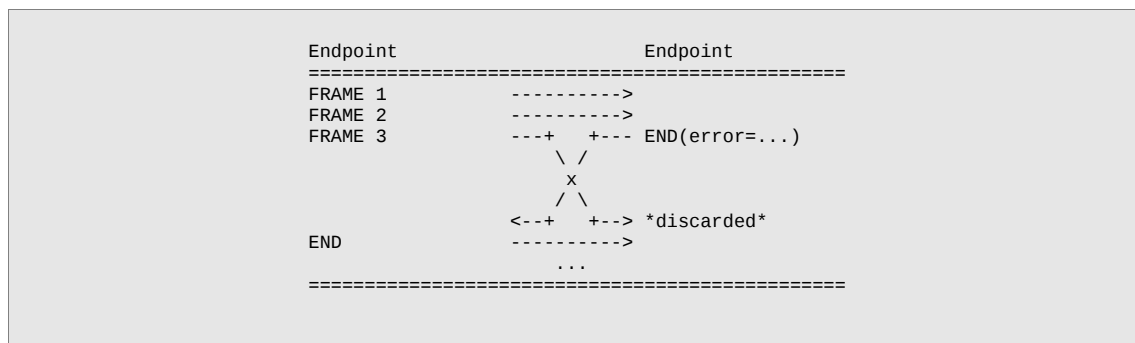
5.3 Simultaneous End

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.



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.



5.5 Session States

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

BEGIN_SENT: In the BEGIN_SENT state, the Session endpoint is assigned an outgoing channel number, but there is no entry in the incoming channel map. In this state the endpoint may send frames but cannot receive them.

BEGIN_RCVD: In the BEGIN_RCVD state, the Session endpoint has an entry in the incoming channel map, but has not yet been assigned an outgoing channel number. The endpoint may receive frames, but cannot send them.

MAPPED: In the MAPPED state, the Session endpoint has both an outgoing channel

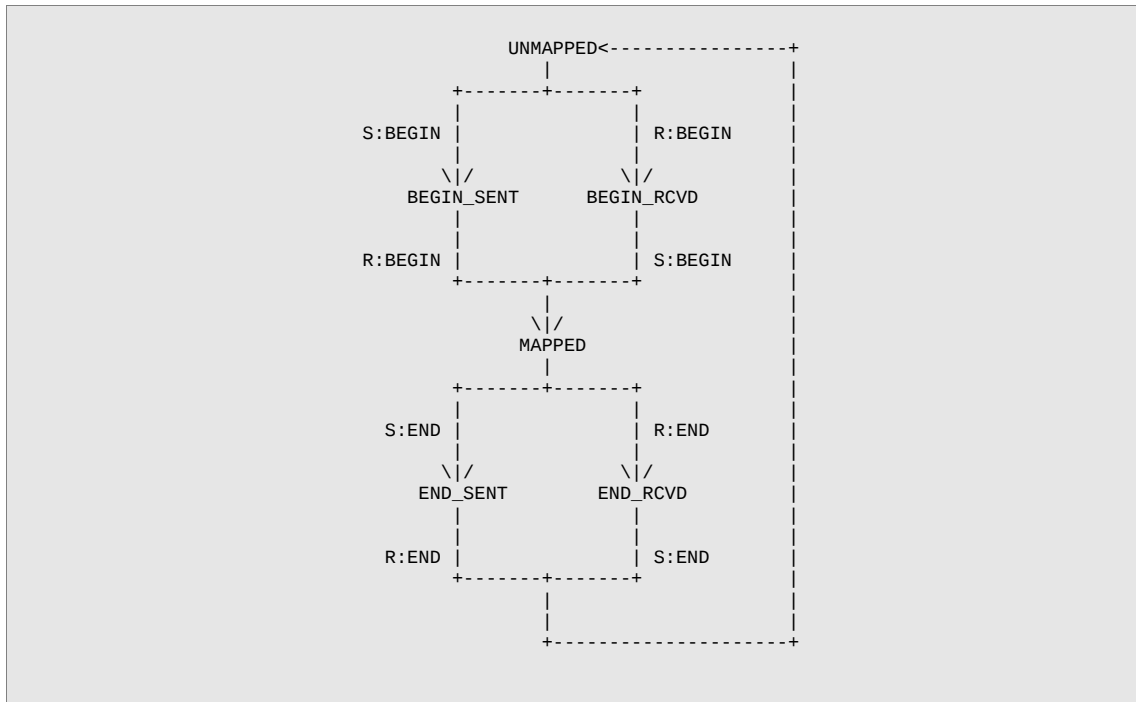
number and an entry in the incoming channel map. The endpoint may both send and receive frames.

END_SENT: In the END_SENT state, the Session endpoint has an entry in the incoming channel map, but is no longer assigned an outgoing channel number. The endpoint may receive frames, but cannot send them.

END_RCVD: In the END_RCVD state, the Session endpoint is assigned an outgoing channel number, but there is no entry in the incoming channel map. The endpoint may send frames, but cannot receive them.

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

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.

6 Links

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

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

Link Endpoints may retain additional state when actively communicating (such as the last known remote transfer state), however this state is not necessary for recovery of a Link. Only the unsettled transfer state is necessary for link recovery, and this need not be stored directly. The form of delivery-tags are intentionally left open-ended so that they and their related transfer state can, if desired, be (re)constructed from application state, thereby minimizing or eliminating the need to retain additional protocol-specific state in order to recover a Link.

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

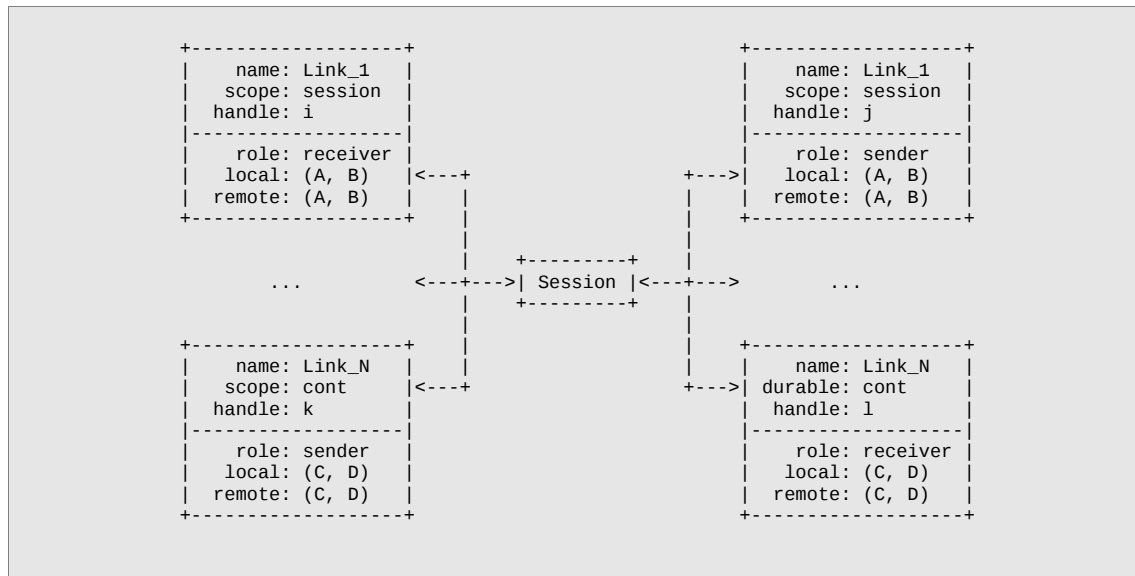
TODO: EXPLAIN SOURCE AND TARGET BETTER AND TALK ABOUT LINKAGE The Sender is associated with a *Source*, and the Receiver is associated with a *Target*. The source defines which Messages are supplied to the Sender for transfer, and the target defines how received Messages are disseminated.

6.1 Naming a Link

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

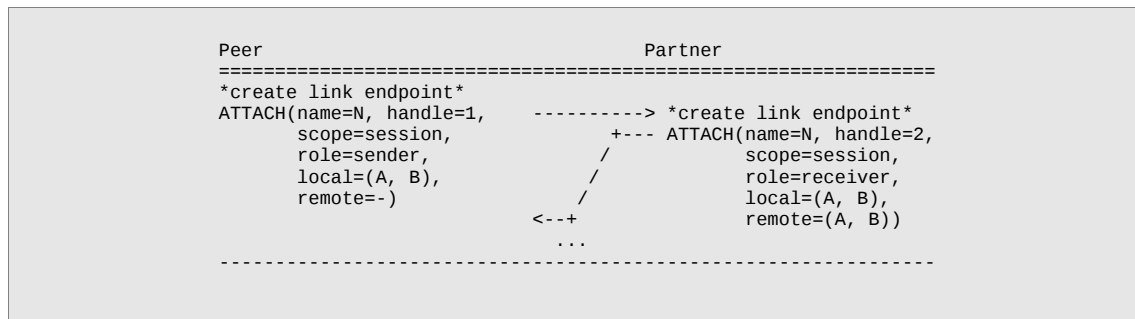
6.2 Link Handles

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

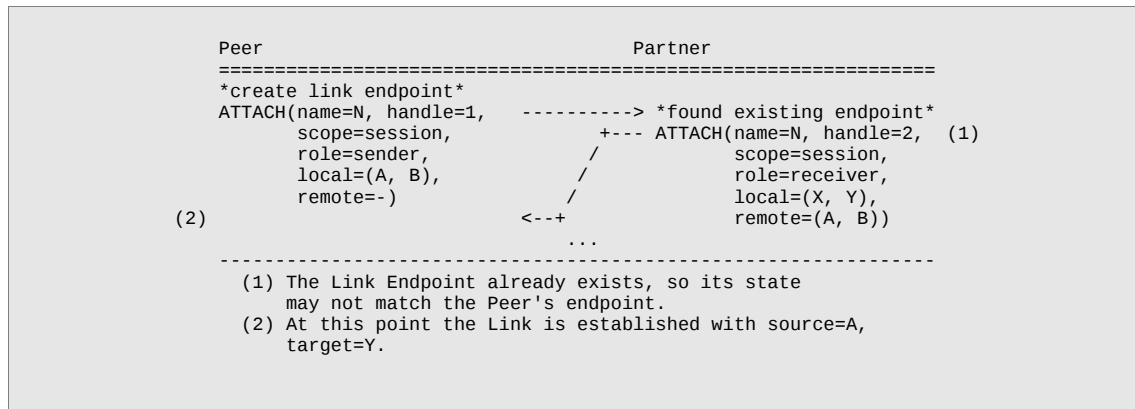


6.3 Establishing a Link

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

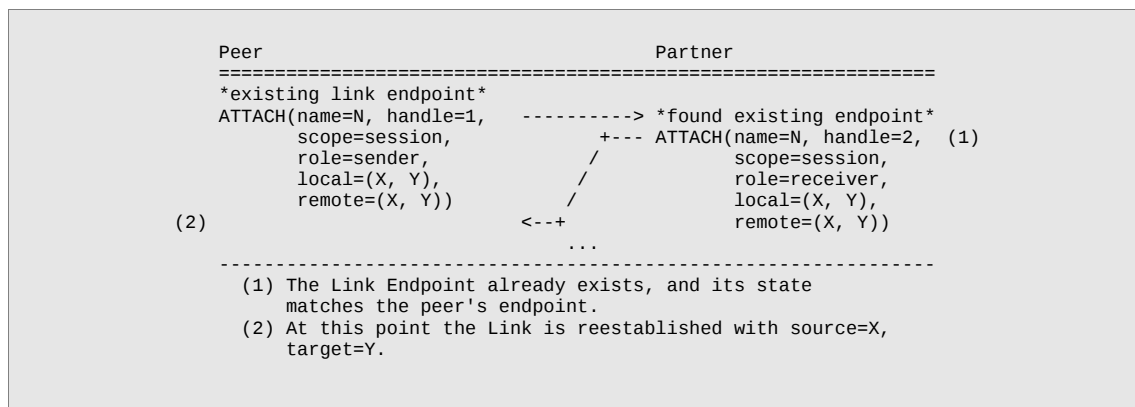


If the partner finds a preexisting Link Endpoint, it **MUST** respond with the state of the existing endpoint rather than creating a new endpoint. This state may not match the state of the peer's endpoint. In the case where a Link is established between two endpoints with conflicting state, the outgoing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

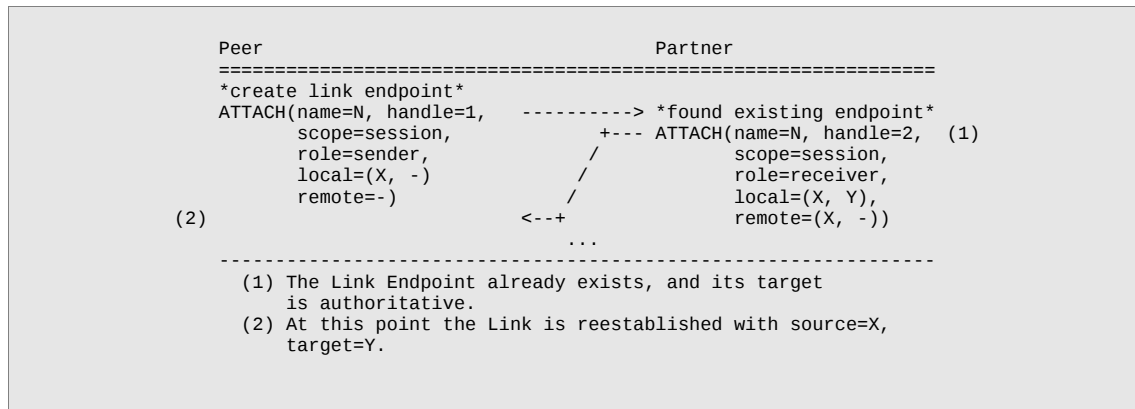


6.4 Resuming a Link

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



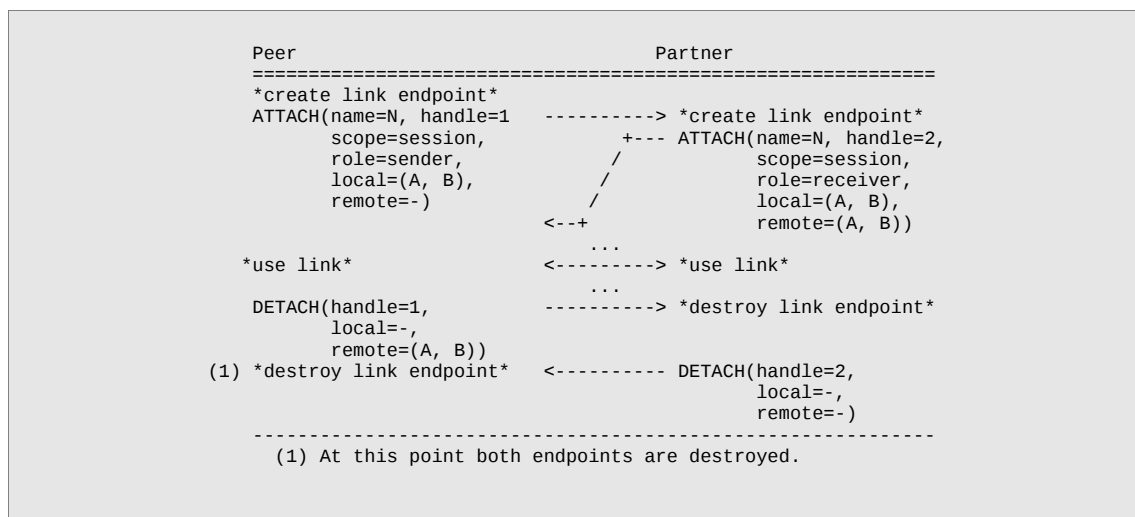
It is possible to resume any Link knowing only the Link name and direction. This is done by creating a new Link Endpoint with an empty source or target for incoming or outgoing Links respectively. The full Link state is then constructed from the authoritative source or target supplied by the other endpoint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



TODO: TALK ABOUT DISASSOCIATING A LINK FROM A SESSION

6.5 Closing a Link

TODO: COVER DETACHING A LINK, AND COVER SIMULTANEOUS DETACH/CLOSE CASE A peer closes a Link by sending the detach frame with the handle for the specified Link, and the local linkage set to null. The partner will destroy the corresponding Link endpoint, and reply with its own detach frame indicating both the local and remote linkage are null.



6.6 Flow Control

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

transfer-count: The *transfer-count* is initialized by the Sender when a link is attached, and is incremented whenever message data is sent or received. The *transfer-unit* defines how the transfer-count is incremented. (See the `attach` definition for details.) Only the Sender may independently modify this field. The

Receiver's value reflects the last-known value indicated by the Sender.

link-credit: The *link-credit* variable defines the current maximum legal amount that the *transfer-count* may be increased. This identifies a *transfer-limit* that may be computed by adding the *link-credit* to the *transfer-count*. If no *link-credit* is set, then there is currently no maximum legal value for the *transfer-count*, i.e. no *transfer-limit*.

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

$$\text{link-creditsnd} := \text{transfer-countrcv} + \text{link-creditrcv} - \text{transfer-countsnd}$$

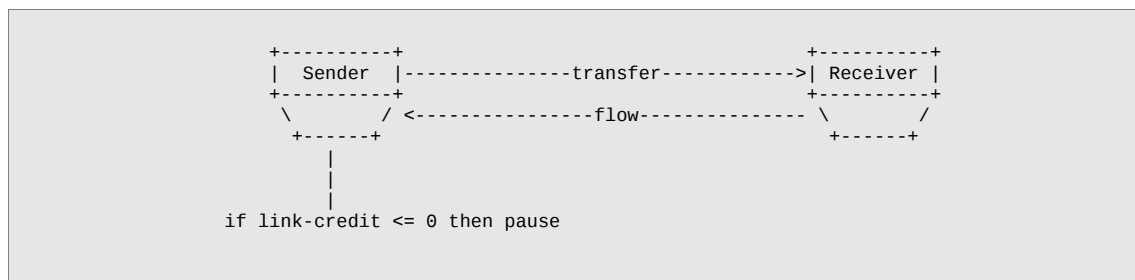
In the event that the receiver does not yet know the *transfer-count*, i.e. *transfer-countrcv* is unspecified, the Sender MUST assume that the *transfer-countrcv* is the first/lowest *transfer-countsnd* sent from Sender to Receiver, i.e. the *transfer-countsnd* specified in the flow state carried by the initial attach frame from the Sender to the Receiver.

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

available: The *available* variable defines the current maximum amount that the *transfer-count* could be increased given sufficient *link-credit*, i.e. the number of available transfer-units at the Sender. Only the Sender can independently modify this field. The Receiver's value is always the last-known value indicated by the Sender.

drain: The drain flag indicates how the Sender should behave when insufficient transfers are available to consume the current *link-credit*. If set, the Sender will (after sending all available transfers) advance the *transfer-count* as much as possible, consuming all *link-credit*, and send the flow-state to the Receiver. Only the Receiver can independently modify this field. The Sender's value is always the last-known value indicated by the Receiver. If *link-credit* is not set (i.e. there is no *transfer-limit* in place), then the value of this flag has no effect.

If the *link-credit* is less than or equal to zero, i.e. the *transfer-count* is the same as or greater than the *transfer-limit*, it is illegal to send more transfers. Flow control may be disabled entirely if the Receiver sends an empty value for *link-credit*. If the *link-credit* is reduced by the Receiver when transfers are in-flight, the Receiver MAY either handle the excess transfers normally or detach the Link with a *transfer-limit-exceeded* error code.



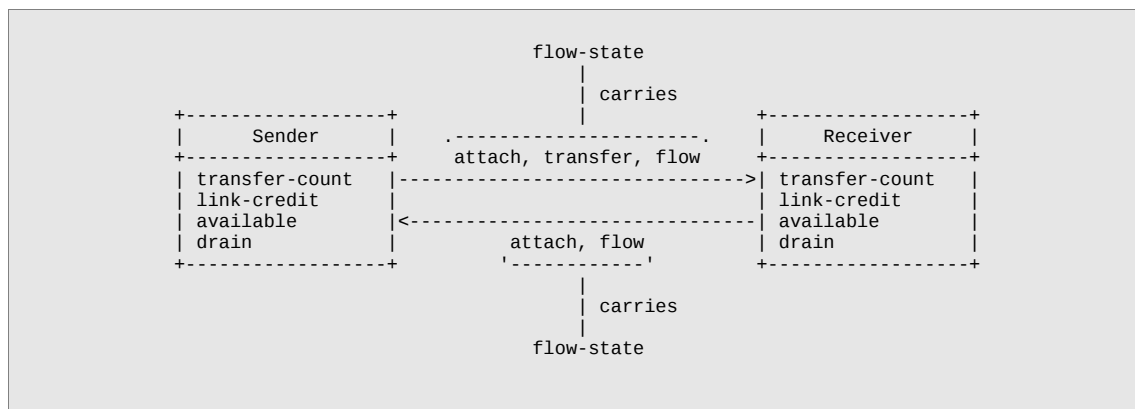
If the Sender's drain flag is set and there are no available transfers, the Sender MUST advance its

transfer-count until link-credit is zero, and send its updated flow-state to the Receiver. **TODO: DIAGRAM?**

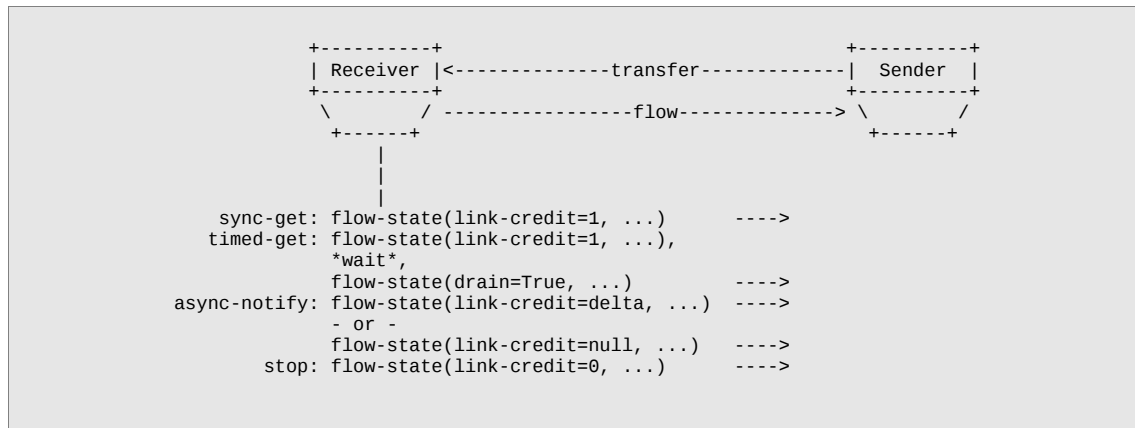
If the sender's link-credit is more than zero, but not sufficient to transfer the next available message, the sender **MUST NOT** assume that more credit will be forthcoming. In particular, if the drain flag is set, the sender **MUST** use all the available credit by fragmenting messages if necessary.

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

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

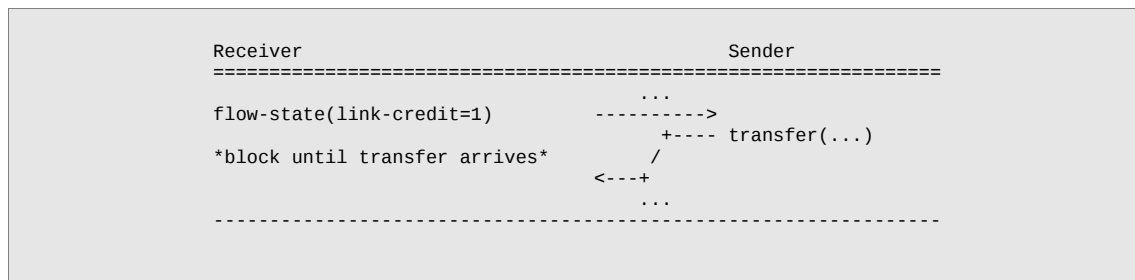


The flow control semantics defined in this section provide the primitives necessary to implement a wide variety of flow control strategies. Additionally, by manipulating the link-credit and drain flag, a Receiver can provide a variety of different higher level behaviors often useful to applications,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

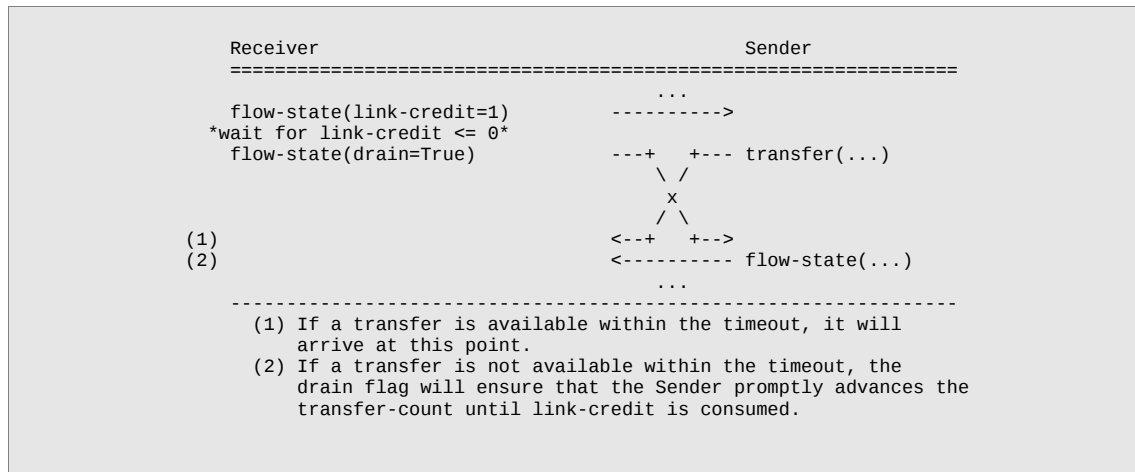


6.7 Synchronous Get

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

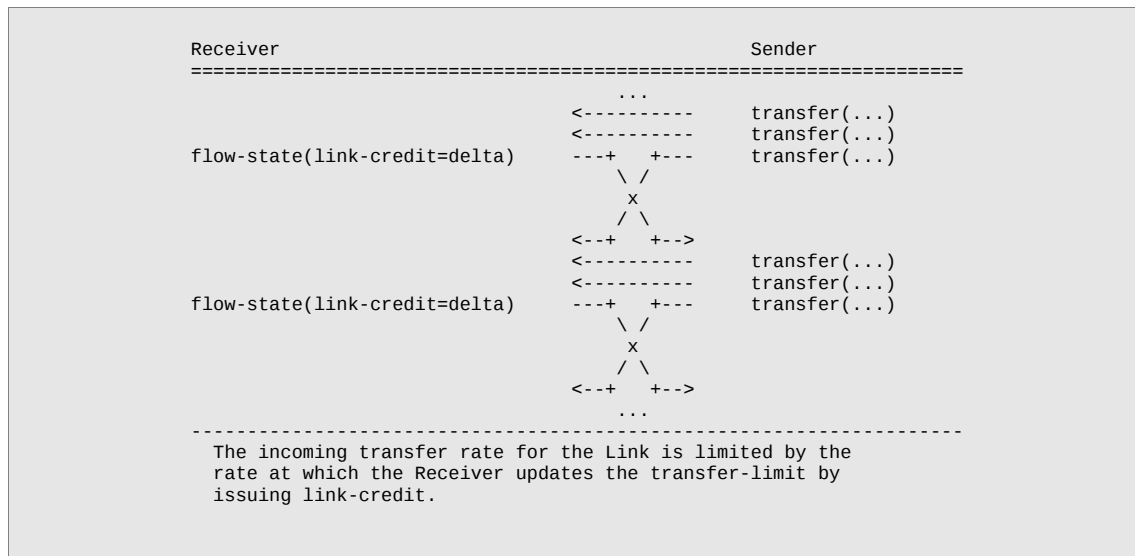


Synchronous get with a timeout is accomplished by incrementing the link-credit, sending the updated flow-state and waiting for the link-credit to be consumed. When the desired time has elapsed the Receiver then sets the drain flag and sends the newly updated flow-state again, while continuing to wait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

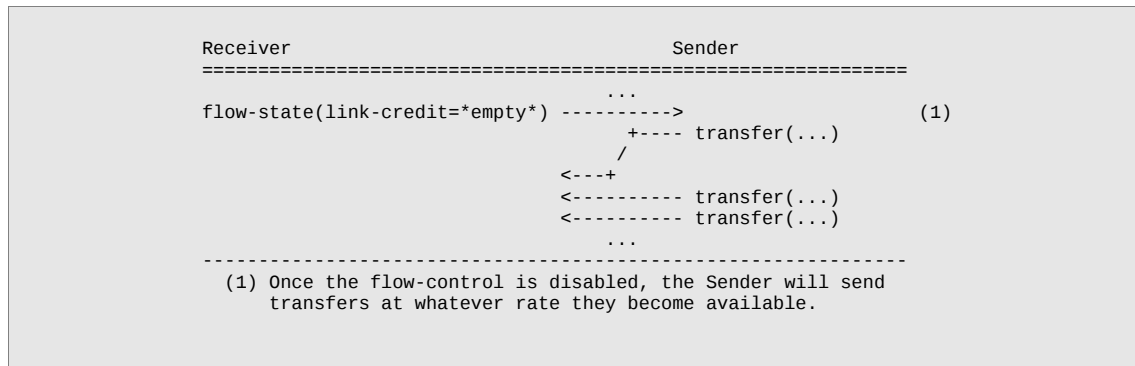


6.8 Asynchronous Notification

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

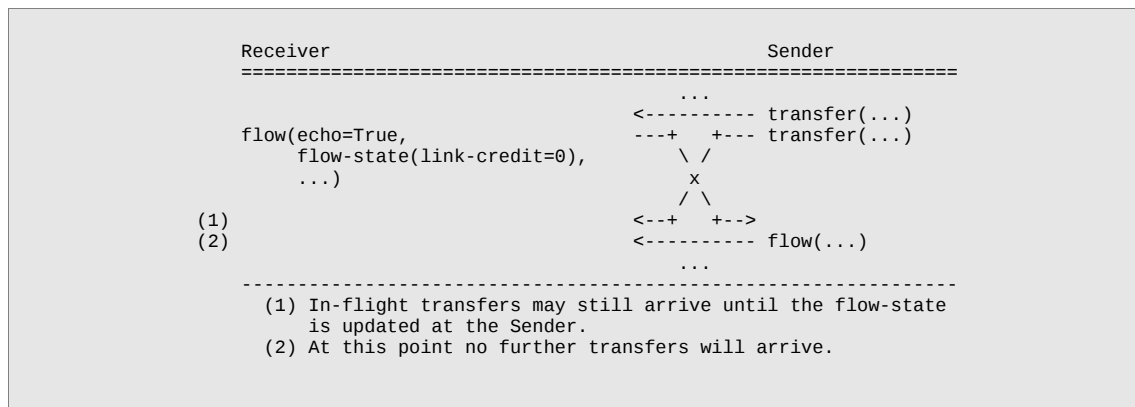


Alternatively, if there is no need to control the rate of incoming transfers for the Link, the receiver can disable flow-control entirely.



6.9 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.



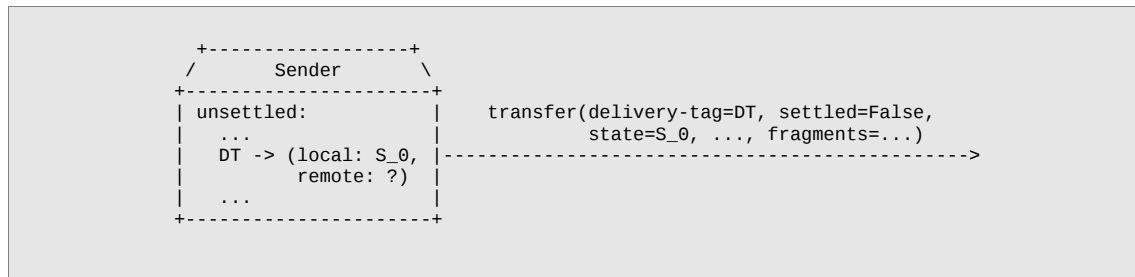
6.10 Transferring a Message

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

The application upon initiating a transfer will supply the sending link endpoint (Sender) with the message data and its associated delivery-tag. The Sender will create an entry in its unsettled map, and send a transfer frame that includes the delivery-tag, its initial state, and its associated message data. For brevity on the wire, the delivery-tag is also associated with a transfer-id assigned by the session. The transfer-id is then used to refer to the delivery-tag in all subsequent interactions on that session. For simplicity the transfer-id is omitted in the following diagrams and the delivery-tag is itself used directly. These diagrams also assume that this interaction takes

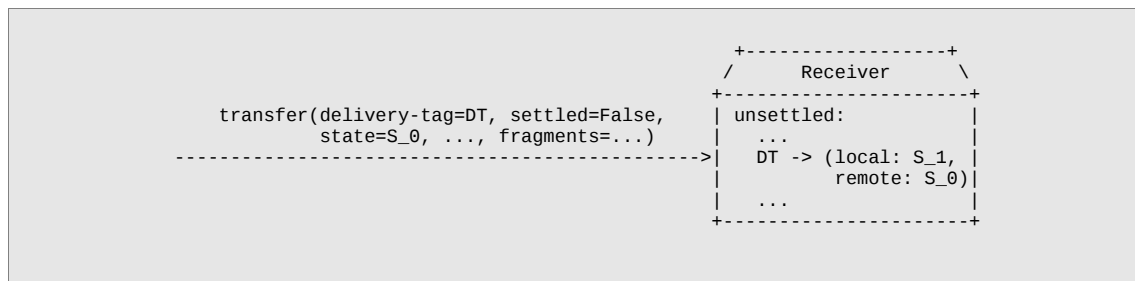
place in the context of a single established link, and as such omit other details that would be present on the wire in practice such as the channel number, link handle, fragmentation flags, etc, focusing only on the essential aspects of message transfer. **TODO: TALK ABOUT WHEN TRANSFER-IDS, FLOW-STATE, ETC, TRANSIENT/SESSION RELATED LINK ENDPOINT STATE IS RETAINED**

Initial Transfer



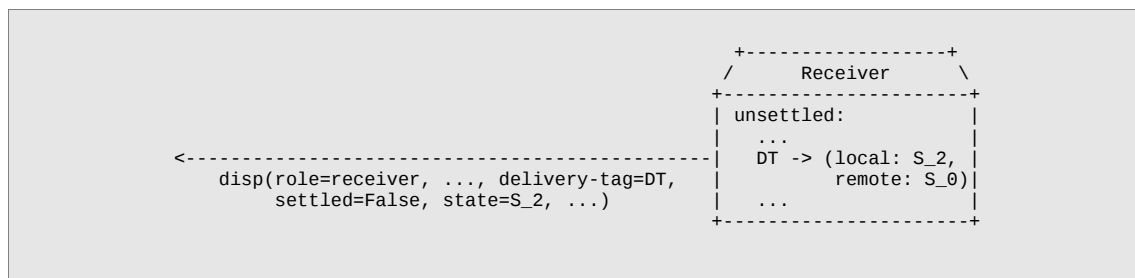
Upon receiving the transfer, the receiving link endpoint (Receiver) will create an entry in its own unsettled map and make the transferred message data available to the application to process.

Initial Receipt



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

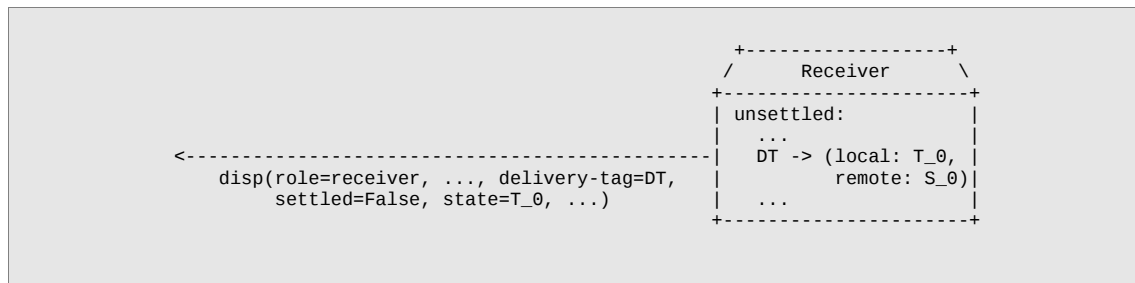
Indication of Non-Terminal State



Once the receiving application has finished processing the message, it indicates to the link endpoint a *terminal* transfer state that reflects the outcome of the application processing

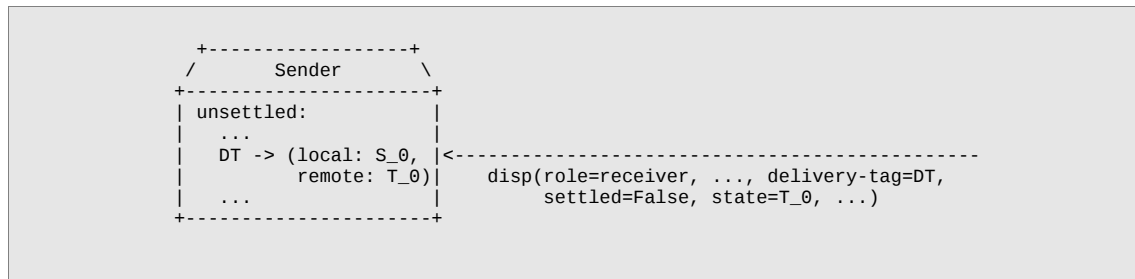
(successful or otherwise). This terminal state is then communicated back to the Sender via the disposition frame.

Indication of Terminal State



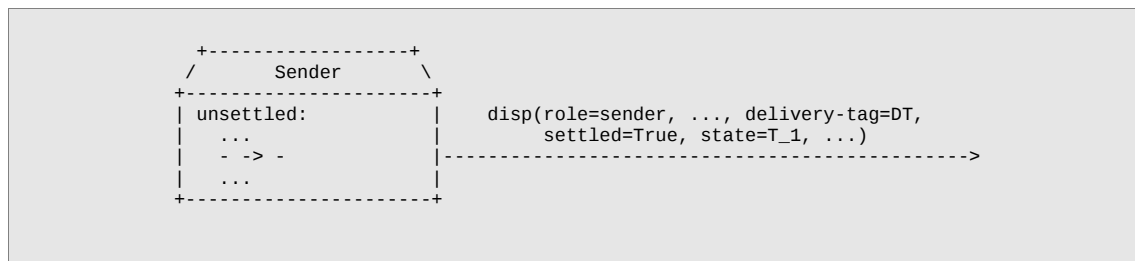
Upon receiving the updated transfer state from the Receiver, the Sender will update its view of the remote state and communicate this back to the sending application.

Receipt of Terminal State



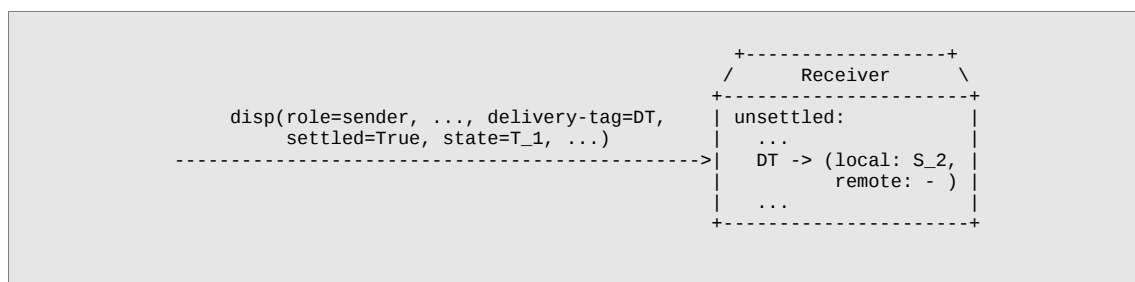
The sending application will then typically perform some action based on this terminal state and then settle the transfer, causing the Sender to remove the delivery-tag from its unsettled map. The Sender will then send its final transfer state along with an indication that the transfer is settled at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

Indication of Settlement



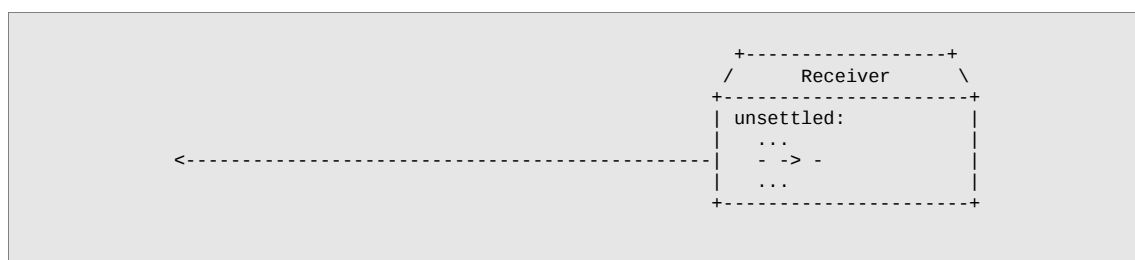
When the Receiver finds out that the Sender has settled the transfer, the Receiver will update its view of the remote state to indicate this, and then notify the receiving application.

Receipt of Settlement



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

Final Settlement

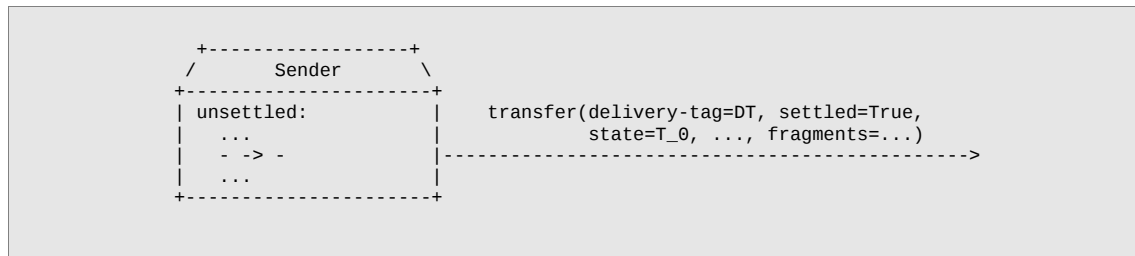


This set of exchanges illustrates the basic principals of message transfer. While a transfer is unsettled the endpoints exchange the current state of the transfer. Eventually both endpoints reach a terminal state as indicated by the application. This triggers the other application to take some final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

This basic pattern can be modified in a variety of ways to achieve different guarantees, for example if the sending application settles the transfer *before* sending it, this results in an *at-most-once* guarantee. The Sender has indicated up front with his initial transmission that he has

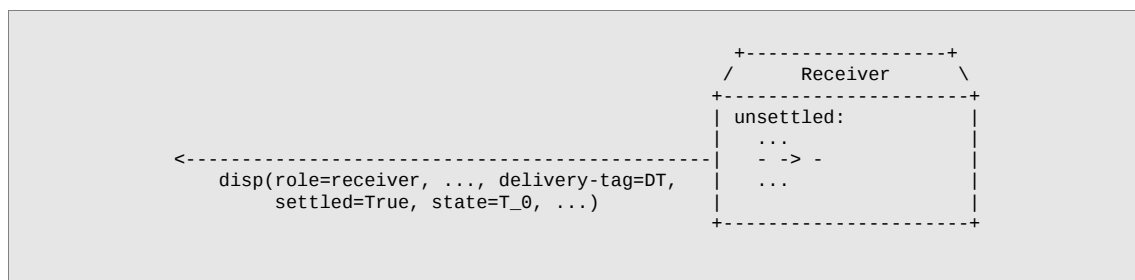
forgotten everything about this transfer and will therefore make no further attempts to send it. Should this transfer make it to the Receiver, the Receiver clearly has no obligation to respond with updates of the Receiver's transfer state, as they would be meaningless and ignored by the Sender.

At-Most-Once



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

At-Least-Once



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

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

TODO: STATE DIAGRAM FOR TRANSFERS

6.11 Resuming Transfers

TODO: DESCRIBE THIS

6.12 Message Fragmentation

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

Section 1		Section 2	
header		body	
F1	F2	F3	F4

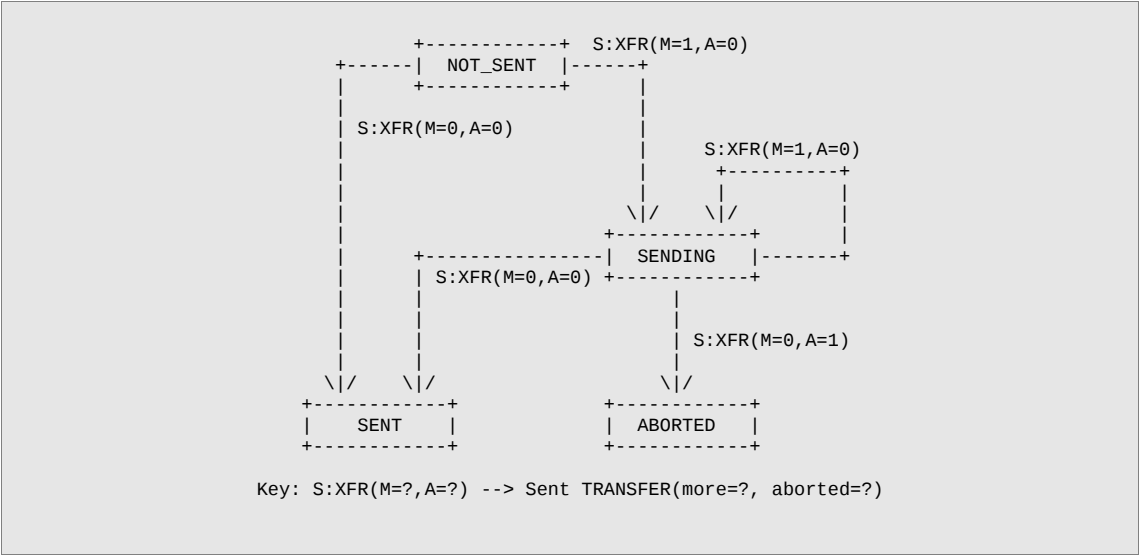
Fragment	first	last	format-code
F1	true	true	0xDB
F2	true	false	0xCA
F3	false	false	0xCA
F4	false	true	0xCA

Each transfer frame may carry an arbitrary number of fragments up to the limit imposed by the maximum frame size. For Messages that are too large to fit within the maximum frame size, additional fragments may be transferred in additional transfer frames by setting the more flag on all but the last transfer frame. When a message is split up into multiple transfer frames in this manner, messages being transferred along different links MAY be interleaved. However, messages transferred along a single link MUST NOT be interleaved.

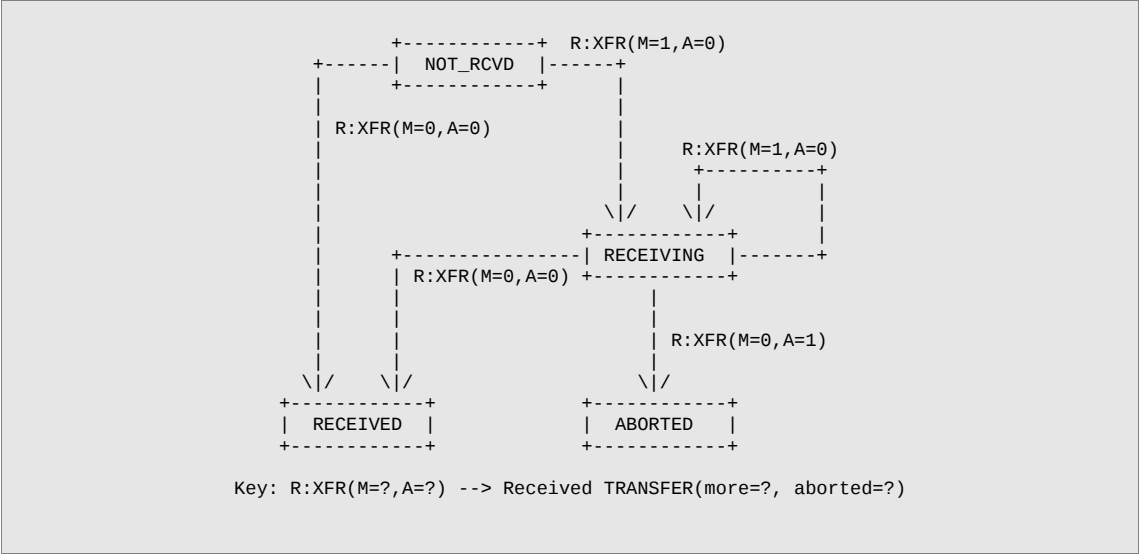
The sender may indicate an aborted attempt to transfer a Message by setting the abort flag on the last transfer. In this case the receiver MUST discard the Message fragments that were transferred prior to the abort.

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

Outgoing Fragmentation State Diagram



Incoming Fragmentation State Diagram



7 Frame-bodies

7.1 open (negotiate Connection parameters)

```
<type class="composite" name="open" source="list" provides="frame">
  <descriptor name="amqp:open:list" code="0x00000001:0x00000100"/>
  <field name="options" type="options"/>
  <field name="container-id" type="string" mandatory="true"/>
  <field name="hostname" type="string"/>
  <field name="max-frame-size" type="uint"/>
  <field name="channel-max" type="ushort" mandatory="true"/>
  <field name="heartbeat-interval" type="seconds"/>
  <field name="outgoing-locales" type="string" multiple="true"/>
  <field name="incoming-locales" type="string" multiple="true"/>
  <field name="offered-capabilities" type="symbol" multiple="true"/>
  <field name="desired-capabilities" type="symbol" multiple="true"/>
  <field name="properties" type="fields"/>
</type>
```

Field Details:

options options map

container-id the id of the source container

hostname the name of the target host

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

max-frame-size proposed maximum frame size

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

heartbeat-interval proposed heartbeat interval

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

outgoing-locales 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. The default is the en_US locale. A peer **MUST** support at least the en_US locale. Since this value is always supported, it need not be supplied in the outgoing-locales.

incoming-locales 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.

offered-capabilities the extension capabilities the sender supports

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

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

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

properties connection properties

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

TODO: REFERENCE EXTERNAL FIELD DEFINITIONS

7.2 begin (begin a Session on a channel)

```
<type class="composite" name="begin" source="list" provides="frame">
  <descriptor name="amqp:begin:list" code="0x00000001:0x00000101"/>
  <field name="options" type="options"/>
  <field name="remote-channel" type="ushort"/>
  <field name="offered-capabilities" type="symbol" multiple="true"/>
  <field name="desired-capabilities" type="symbol" multiple="true"/>
  <field name="properties" type="fields"/>
</type>
```

Field Details:

options options map

remote-channel the remote channel for this Session

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

offered-capabilities the extension capabilities the sender supports

desired-capabilities	the extension capabilities the sender may use if the receiver supports them
properties	session properties

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

TODO: REFERENCE EXTERNAL FIELD DEFINITIONS

7.3 attach (attach a Link to a Session)

```
<type class="composite" name="attach" source="list" provides="frame">
  <descriptor name="amqp:attach:list" code="0x00000001:0x00000102"/>
  <field name="options" type="options"/>
  <field name="name" type="string" mandatory="true"/>
  <field name="handle" type="handle" mandatory="true">
    <error name="handle-busy" type="amqp-error" value="invalid-field"/>
  </field>
  <field name="flow-state" type="flow-state" mandatory="true"/>
  <field name="role" type="role" mandatory="true"/>
  <field name="local" type="linkage"/>
  <field name="remote" type="linkage"/>
  <field name="durable" type="boolean"/>
  <field name="expiry-policy" type="link-expiry-policy" default="session"/>
  <field name="timeout" type="seconds" default="0"/>
  <field name="unsettled" type="map"/>
  <field name="transfer-unit" type="ulong"/>
  <field name="max-message-size" type="ulong"/>
  <field name="error-mode" type="error-mode"/>
  <field name="properties" type="fields"/>
</type>
```

Field Details:

options	options map
name	the name of the link
<p>This name uniquely identifies the link from the container of the source to the container of the target node, e.g. if the container of the source node is A, and the container of the target node is B, the link may be globally identified by the compound name <i>A:B:name</i>. TODO: ADD EXAMPLES/PICTURES TO MAKE THIS CLEAR. If the remote Link Endpoint does not exist, the peer MUST create a new one or end the session with an error.</p>	
role	role of the link endpoint
local	the local source and target definitions
remote	the remote source and target definitions
durable	indicates that the unassociated Link Endpoint will be durably retained
expiry-policy	the expiry policy of the link
<p>The value of this field determines when the expiry timer controlled by the timeout field starts counting down.</p>	
timeout	duration that an expiring Link Endpoint will be retained
<p>A Link Endpoint starts expiring as indicated by the expiry-policy.</p>	
unsettled	unsettled transfer state

transfer-unit the transfer unit

The transfer-unit need only be set by the incoming link endpoint. Any value set by an outgoing link endpoint is ignored.

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

properties link properties

TODO: REFERENCE EXTERNAL FIELD DEFINITIONS

```
<type class="composite" name="flow" source="list" provides="frame">
  <descriptor name="amqp:flow:list" code="0x00000001:0x00000103"/>
  <field name="options" type="options"/>
  <field name="handle" type="handle" mandatory="true">
    <error name="unattached-handle" type="amqp-error" value="invalid-field"/>
  </field>
  <field name="flow-state" type="flow-state" mandatory="true"/>
  <field name="echo" type="boolean"/>
</type>
```

echo

request link state from other endpoint

7.5 transfer (transfer a Message)

```
<type class="composite" name="transfer" source="list" provides="frame">
  <descriptor name="amqp:transfer:list" code="0x00000001:0x00000104"/>
  <field name="options" type="options"/>
  <field name="handle" type="handle" mandatory="true">
    <error name="unattached-handle" type="amqp-error" value="invalid-field"/>
  </field>
  <field name="flow-state" type="flow-state" mandatory="true"/>
  <field name="delivery-tag" type="delivery-tag" mandatory="true"/>
  <field name="transfer-id" type="transfer-number" mandatory="true"/>
  <field name="settled" type="boolean"/>
  <field name="state" type=""/>
  <field name="resume" type="boolean"/>
  <field name="more" type="boolean"/>
  <field name="aborted" type="boolean"/>
  <field name="batchable" type="boolean"/>
  <field name="fragments" type="fragment" multiple="true"/>
</type>
```

Field Details:

options

options map

handle

Specifies the Link on which the Message is transferred.

delivery-tag

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

transfer-id

alias for delivery-tag

more

indicates that the Message has more content

aborted

indicates that the Message is aborted

Aborted Messages should be discarded by the recipient.

batchable

TODO: doc

7.6 disposition (inform remote peer of transfer state changes)

```
<type class="composite" name="disposition" source="list" provides="frame">
  <descriptor name="amqp:disposition:list" code="0x00000001:0x00000105"/>
  <field name="options" type="options"/>
  <field name="role" type="role" mandatory="true"/>
  <field name="batchable" type="boolean"/>
  <field name="extents" type="extent" multiple="true"/>
</type>
```

Field Details:

7.7 detach (detach the Link Endpoint from the Session)

```
<type class="composite" name="detach" source="list" provides="frame">
  <descriptor name="amqp:detach:list" code="0x00000001:0x00000106"/>
  <field name="options" type="options"/>
</type>
```

```

<field name="handle" type="handle" mandatory="true"/>
<field name="local" type="linkage"/>
<field name="remote" type="linkage"/>
<field name="error" type="error"/>
</type>

```

Field Details:

options	options map
local	the local source and target definitions
remote	the remote source and target definitions
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.

7.8 end (end the Session)

```

<type class="composite" name="end" source="list" provides="frame">
  <descriptor name="amqp:end:list" code="0x00000001:0x00000107"/>
  <field name="options" type="options"/>
  <field name="error" type="error"/>
</type>

```

Field Details:

options	options map
error	error causing the end

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

7.9 close (signal a Connection close)

```

<type class="composite" name="close" source="list" provides="frame">
  <descriptor name="amqp:close:list" code="0x00000001:0x00000108"/>
  <field name="options" type="options"/>
  <field name="error" type="error"/>
</type>

```

Field Details:

options	options map
error	error causing the close

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

8 Definitions

8.1 *role: boolean(link endpoint role)*

Valid values:

false (sender)
true (receiver)

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

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

8.3 *linkage (the source and target for a link)*

```
<type class="composite" name="linkage" source="list">  
  <descriptor name="amqp:linkage:list" code="0x00000001:0x00000109"/>  
  <field name="source" type="*" requires="source"/>  
  <field name="target" type="*" requires="target"/>  
</type>
```

Field Details:

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.

8.4 *link-expiry-policy: symbol(expiry policy for a link endpoint)*

Valid values:

session Expiry starts when the link is detached and the last associated session is closed.
connection Expiry starts counting when the link is detached and the last associated connection is closed.
never The link never expires.

8.5 seconds: *uint(a duration measured in seconds)*

8.6 flow-state

```
<type class="composite" name="flow-state" source="list">
  <descriptor name="amqp:flow-state:list" code="0x00000001:0x0000010a"/>
  <field name="unsettled-lwm" type="transfer-number"/>
  <field name="session-credit" type="uint" mandatory="true"/>
  <field name="transfer-count" type="sequence-no"/>
  <field name="link-credit" type="uint"/>
  <field name="available" type="uint"/>
  <field name="drain" type="boolean"/>
</type>
```

Field Details:

unsettled-lwm

low-water mark of unsettled transfers

TODO: DOCUMENT THIS FIELD AND SESSION-CREDIT. IN PARTICULAR THIS FIELD MAY ONLY BE NULL UNTIL THE SESSION ENDPOINT BECOMES AWARE OF THE FIRST TRANSFER-ID SENT BY ITS PEER. ALSO NOTE THAT IF ALL OUTSTANDING TRANSFERS ARE SETTLED THIS FIELD WILL REFERENCE THE NEXT EXPECTED TRANSFER. FURTHER NOTE THAT THIS AND SESSION-CREDIT ARE REFERENCING THE SENDING STATE IF THE FLOW STATE IS REFERRING TO A SENDING LINK ENDPOINT, AND THE RECEIVING SESSION STATE IF THIS STATE IS REFERRING TO A RECEIVING LINK ENDPOINT.

session-credit

the number of transfers that can be sent/received without settling

TODO: DOC

transfer-count

the endpoint's transfer-count

This field contains the current transfer-count of the endpoint. This field **MUST** always be set by the Sender. This field **MUST** always be set by the Receiver to the last known Sender's value. This field is empty if and only if the Receiver has not yet seen the initial attach frame from the Sender.

link-credit

the current maximum legal transfer-count

TODO: DOC

available

the number of available transfer units

TODO: DOC

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.

8.7 delivery-tag: *binary*

8.8 transfer-number: *sequence-no*

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

A sequence-no encodes a serial number as defined in RFC-1982. The arithmetic, and operators

for these numbers are defined by RFC-1982.

8.10 extent

```
<type name="extent" class="composite" source="list">
  <descriptor name="amqp:extent:list" code="0x00000001:0x0000010b"/>
  <field name="first" type="transfer-number" mandatory="true"/>
  <field name="last" type="transfer-number"/>
  <field name="handle" type="uint"/>
  <field name="settled" type="boolean"/>
  <field name="state" type="*" requires="transfer-state"/>
</type>
```

Field Details:

8.11 fragment (a Message fragment)

```
<type class="composite" name="fragment" source="list">
  <descriptor name="amqp:fragment:list" code="0x00000001:0x0000010c"/>
  <field name="first" type="boolean"/>
  <field name="last" type="boolean"/>
  <field name="format-code" type="uint"/>
  <field name="fragment-offset" type="ulong"/>
  <field name="payload" type="binary"/>
</type>
```

Field Details:

first	indicates the fragment is the first in the Section
If this flag is true, then the beginning of the payload corresponds with a section boundary within the Message.	
last	indicates the fragment is the last in the Section
If this flag is true, then the end of the payload corresponds with a section boundary within the Message.	
format-code	indicates the format of the Message section
The format code indicates the format of the current section of the Message. A Message may have multiple sections, and therefore multiple format codes, however the format code is only permitted to change at section boundaries.	
fragment-offset	the payload offset within the Message
payload	Message data

8.12 fields: *map(a mapping from field name to value)*

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

8.13 options: *fields(options map)*

The options map is used to convey domain or vendor specific optional modifiers on defined

AMQP types, e.g. frame bodies. Each key in the map must be of the type `symbol` and all keys except those beginning with the string "x-" are reserved.

On receiving a type with an options map containing keys or values which it does not recognise, and for which the key does not begin with the string "x-opt-" an AMQP container **MUST** indicate an error.

8.14 error-mode: *symbol*(behaviors for link errors)

Valid values:

- `detach` When a link error is encountered, the link is detached.
- `end` When a link error is encountered, the link is detached and immediately following this the session is ended with an error code of `session-error`.

8.15 error (details of an error)

```
<type class="composite" name="error" source="list">
  <descriptor name="amqp:error:list" code="0x00000001:0x0000010d"/>
  <field name="condition" type="symbol" requires="error" mandatory="true"/>
  <field name="description" type="string"/>
  <field name="info" type="map"/>
</type>
```

Field Details:

condition	error condition
	A symbolic value indicating the error condition.
description	descriptive text about the error condition
	This text supplies any supplementary details not indicated by the condition field. This text can be logged as an aid to resolving issues.
info	map carrying information about the error condition

8.16 amqp-error: *symbol(shared error conditions)*

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.

8.17 connection-error: *symbol(symbols used to indicate connection error conditions)*

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.

8.18 session-error: *symbol(symbols used to indicate session error conditions)*

Valid values:

amqp:session:unsettled-limit-exceeded	The peer exceeded the unsettled-limit for the session.
amqp:session:link-error	A link with an error-mode of "end" encountered an error.

8.19 link-error: *symbol(symbols used to indicate link error conditions)*

Valid values:

- amqp:link:detach-forced An operator intervened to detach for some reason.
- amqp:link:transfer-limit-exceeded The peer sent more Message transfers than currently allowed on the session.
- amqp:link:message-size-exceeded The peer sent a larger message than is supported on the link.

Definition: PORT

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

The standard AMQP port number that has been assigned by IANA for TCP, UDP, and SCTP. There is currently no UDP mapping defined for AMQP. The UDP port number is reserved for future transport mappings.

Definition: MAJOR

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

Definition: MINOR

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

Definition: REVISION

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

Definition: MIN-MAX-FRAME-SIZE

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

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

until they have agreed a definitive maximum frame size for that Connection. **TODO:** EXPLAIN CLEARLY WHAT THIS MEANS AND POSSIBLY RENAME

TODO: GO THROUGH TRIAGE.TXT

TODO: EXPLAIN FRAGMENTATION

TODO: EXPLAIN REQUIREMENTS OF TYPE SYSTEM, I.E. BROKERS NEED TO BE ABLE TO DO USEFUL OPERATIONS WITH NO KNOWLEDGE OF ENDPOINT SCHEMAS

TODO: MAKE IT CLEAR THAT NODES ARE ABSTRACT

Book IV - Messaging

1 Introduction

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

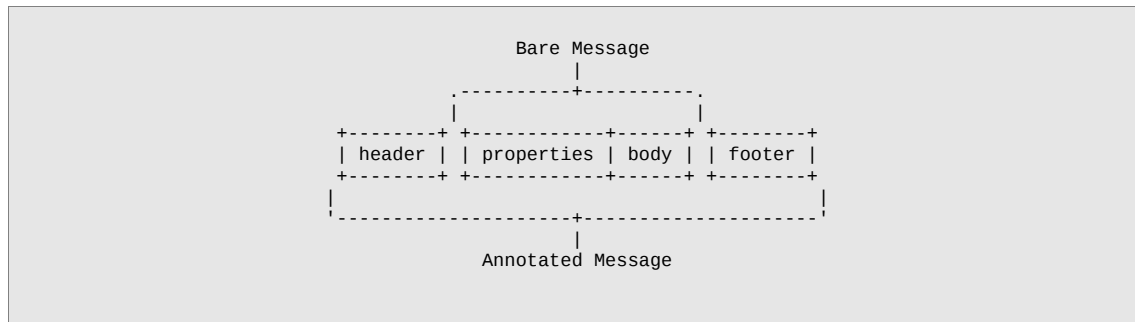
- message format
 - properties for the bare message
 - formats for structured and unstructured sections in the bare message
 - headers and footers for the annotated message
- transfer states for messages traveling between nodes
- distribution nodes
 - states for messages stored at a distribution node
- 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

2 Message Format

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

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

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

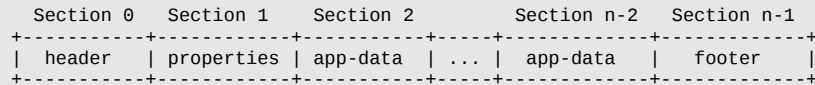


The bare message is immutable within the AMQP network. That is neither the opaque body, nor the properties can be changed by any node acting as an AMQP intermediary. **TODO: INTRODUCE THE CONCEPT OF AN AMQP INTERMEDIARY SOMEWHERE BEFORE HERE**

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

Altogether the message consists of the following sections:

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



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

2.1 section-codes: *uint*

Valid values:

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

2.2 header (transport headers for a Message)

```
<type class="composite" name="header" source="list">
  <descriptor name="amqp:header:list" code="0x00000001:0x00000200"/>
  <field name="durable" type="boolean"/>
  <field name="priority" type="ubyte"/>
  <field name="transmit-time" type="timestamp"/>
  <field name="ttl" type="ulong"/>
  <field name="former-acquirers" type="uint"/>
  <field name="delivery-failures" type="uint"/>
  <field name="format-code" type="uint"/>
  <field name="message-attrs" type="message-attributes"/>
  <field name="delivery-attrs" type="message-attributes"/>
</type>
```

Field Details:

`durable` specify durability requirements

Durable Messages **MUST NOT** be lost even if an intermediary is unexpectedly

terminated and restarted.

priority

relative Message priority

This field contains the relative Message priority. Higher numbers indicate higher priority Messages. Messages with higher priorities MAY be delivered before those with lower priorities.

An AMQP intermediary implementing distinct priority levels MUST do so in the following manner:

- If n distinct priorities are implemented and n is less than 10 -- priorities 0 to $(5 - \text{ceiling}(n/2))$ MUST be treated equivalently and MUST be the lowest effective priority. The priorities $(4 + \text{floor}(n/2))$ and above MUST be treated equivalently and MUST be the highest effective priority. The priorities $(5 - \text{ceiling}(n/2))$ to $(4 + \text{floor}(n/2))$ 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 implemented the 0 to 3 are equivalent, 5 to 9 are equivalent and 3, 4 and 5 are distinct.

This scheme ensures that if two priorities are distinct for a server which implements m separate priority levels they are also distinct for a server which implements n different priority levels where $n > m$.

transmit-time

the time of Message transmit

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

ttl

time to live in ms

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

former-acquirers

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

delivery-failures

the number of prior unsuccessful delivery attempts

The number of unsuccessful previous attempts to deliver this message. If this value is non-zero it may be taken as an indication that the Message may be a duplicate. The `delivery-failures` value is initially set to the same value as the Message has when it arrived at the source. It is incremented upon an outcome being settled at the

sender, according to rules defined for each outcome.

format-code	indicates the format of the Message
message-attrs	message attributes

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

delivery-attrs	delivery attributes
----------------	---------------------

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

2.3 properties (immutable properties of the Message)

```
<type class="composite" name="properties" source="list">
  <descriptor name="amqp:properties:list" code="0x00000001:0x00000201"/>
  <field name="message-id" type="binary"/>
  <field name="user-id" type="binary"/>
  <field name="to" type="*" requires="address"/>
  <field name="subject" type="string"/>
  <field name="reply-to" type="*" requires="address"/>
  <field name="correlation-id" type="binary"/>
  <field name="content-length" type="ulong"/>
  <field name="content-type" type="symbol"/>
</type>
```

Field Details:

message-id	application Message identifier
------------	--------------------------------

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

user-id	creating user id
---------	------------------

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

to	the address of the Node the Message is destined for
----	---

The to field identifies the Node that is the intended destination of the Message. On any given transfer this may not be the Node at the receiving end of the Link.

subject	the subject of the message
---------	----------------------------

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

reply-to	the Node to send replies to
The address of the Node to send replies to.	
correlation-id	application correlation identifier
This is a client-specific id that may be used to mark or identify Messages between clients. The server ignores this field.	
content-length	length of the combined payload in bytes
The total size in octets of the combined payload of all fragment that together make the Message.	
content-type	MIME content type
The RFC-2046 MIME type for the Message content (such as "text/plain"). This is set by the originating client. As per RFC-2046 this may contain a charset parameter defining the character encoding used: e.g. 'text/plain; charset="utf-8"'.	

2.4 footer (transport footers for a Message)

```
<type class="composite" name="footer" source="list">
  <descriptor name="amqp:footer:list" code="0x00000001:0x00000202"/>
  <field name="message-attrs" type="message-attributes"/>
  <field name="delivery-attrs" type="message-attributes"/>
</type>
```

Field Details:

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

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

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

3 Transfer State

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

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

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

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

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

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

```
<type class="composite" name="transfer-state" provides="transfer-state" source="map">
  <descriptor name="amqp:transfer-state:map" code="0x00000001:0x00000203"/>
  <field name="bytes-transferred" type="ulong"/>
  <field name="outcome" type="*" requires="outcome"/>
  <field name="txn-id" type="*" />
</type>
```

Field Details:

3.2 accepted (the accepted outcome)

```
<type class="composite" name="accepted" source="map" provides="outcome"/>
```

3.3 rejected (the rejected outcome)

```
<type class="composite" name="rejected" source="map" provides="outcome">
  <descriptor name="amqp:rejected:map" code="0x00000001:0x00000205"/>
  <field name="error" type="error"/>
</type>
```

Field Details:

error

error that caused the message to be rejected

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

3.4 released (the released outcome)

```
<type class="composite" name="released" source="map" provides="outcome"/>
```

3.5 modified (the modified outcome)

```
<type class="composite" name="modified" source="map" provides="outcome">
  <descriptor name="amqp:modified:map" code="0x00000001:0x00000207"/>
  <field name="delivery-failed" type="boolean"/>
  <field name="deliver-elsewhere" type="boolean"/>
  <field name="message-attrs" type="message-attributes"/>
  <field name="delivery-attrs" type="message-attributes"/>
</type>
```

Field Details:

delivery-failed

count the transfer as an unsuccessful delivery attempt

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

deliver-elsewhere

prevent redelivery

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

message-attrs

message attributes

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

delivery-attrs

delivery attributes

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

4 Distribution Nodes

TODO: FORMALLY INTRODUCE DISTRIBUTION NODES

4.1 Message States

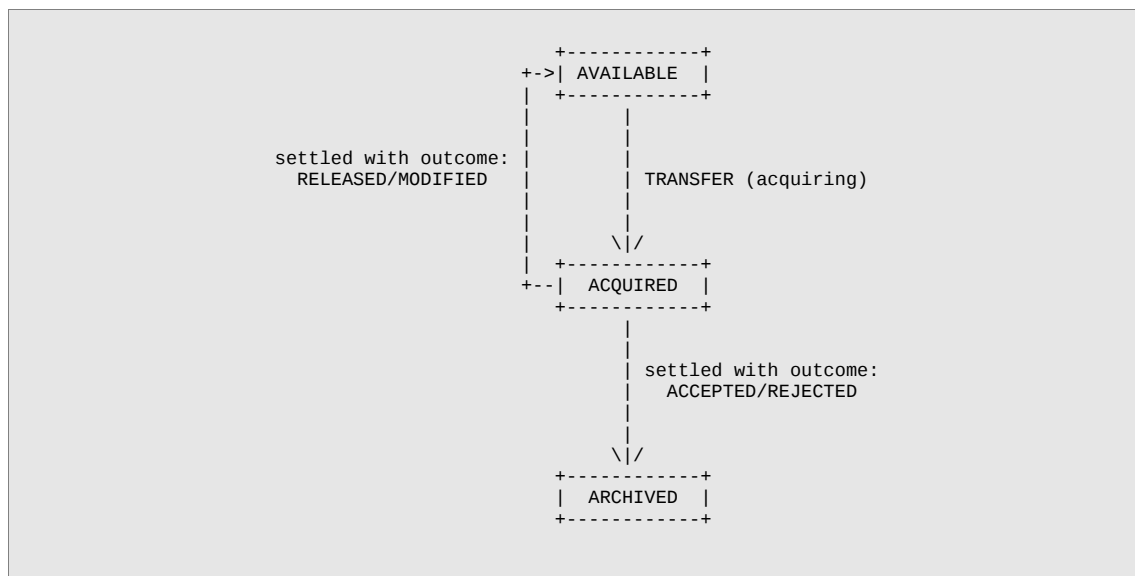
The Messaging layer defines a set of states for Messages stored at a *distribution node*. Not all Nodes store Messages for distribution, however these definitions permit some standardized interaction with those nodes that do. The transitions between these states are controlled by the transfer of Messages to/from a distribution node and the resulting terminal transfer state. Note that the state of a Message at one distribution node does not affect the state of the same Message at a separate node.

By default a Message will begin in the AVAILABLE state. Prior to initiating an *acquiring* transfer, the Message will transition to the ACQUIRED state. Once in the ACQUIRED state, a Messages is ineligible for *acquiring* transfers to any other Links.

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

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

Message State Transitions



5 Sources and Targets

TODO: THIS WHOLE SECTION NEEDS MORE/BETTER EXPLANATION

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

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

5.1 Filtering Messages

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

5.2 Distribution Modes

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

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

A Source MUST NOT resend a Message which has previously been successfully transferred from the Source, i.e. reached an ACCEPTED transfer state at the receiver. For a *move* link with a default configuration this is trivially achieved as such an end result will lead to the Message in the ARCHIVED state on the Node, and thus anyway ineligible for transfer. For a *copy* link, state must be retained at the source to ensure compliance. In practice, for nodes which maintain a strict order on Messages at the node, the state may simply be a record of the most recent Message transferred.

5.3 source

```
<type class="composite" name="source" provides="source" source="map">
  <descriptor name="amqp:source:map" code="0x00000001:0x00000208"/>
```

```

<field name="address" type="*" requires="address">
  <error name="address-not-found" type="amqp-error" value="not-found"/>
</field>
<field name="dynamic" type="*" requires="lifetime-policy"/>
<field name="distribution-mode" type="symbol" requires="distribution-mode"/>
<field name="filter" type="filter-set"/>
<field name="default-outcome" type="*" />
<field name="outcomes" type="symbol" multiple="true"/>
<field name="capabilities" type="symbol" multiple="true"/>
</type>

```

Field Details:

address

the address of the source

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

dynamic

request dynamic creation of a remote Node

When set in the local Linkage by the receiving Link endpoint, this field constitutes a request for the sending peer to dynamically create a Node at the source. In this case the address field **MUST NOT** be set.

When set in the local Linkage by the sending Link Endpoint this field indicates the lifetime policy in use for the Node which has been dynamically created at the time of the `attach`. In this case the address field will contain the address of the created Node. The generated address **SHOULD** include the Link name and Session-name or client-id in some recognizable form for ease of traceability.

The lifetime of the generated node is controlled by the policy specified in the `dynamic` field in the Source of the local Linkage of the sending Link Endpoint. Definitionally, the lifetime will never be less than the lifetime of the link which caused its creation.

distribution-mode

the distribution mode of the Link

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

filter

a set of predicate to filter the Messages admitted onto the Link

default-outcome

default outcome for unsettled transfers

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

outcomes

descriptors for the outcomes that can be chosen on this link

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

capabilities

the extension capabilities the sender supports/desires

5.4 target

```
<type class="composite" name="target" provides="target" source="map">
  <descriptor name="amqp:target:map" code="0x00000001:0x00000209"/>
  <field name="address" type="*" requires="address">
    <error name="address-not-found" type="amqp-error" value="not-found"/>
  </field>
  <field name="dynamic" type="*" requires="lifetime-policy"/>
  <field name="capabilities" type="symbol" multiple="true"/>
</type>
```

Field Details:

address

The address of the target.

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

dynamic

request dynamic creation of a remote Node

When set in the local Linkage by the sending Link endpoint, this field constitutes a request for the receiving peer to dynamically create a Node at the target. In this case the address field **MUST NOT** be set.

When set in the local Linkage by the receiving Link Endpoint this field indicates the lifetime policy in use for the Node which has been dynamically created at the time of the attach. In this case the address field will contain the address of the created Node. The generated address **SHOULD** include the Link name and Session-name or client-id in some recognizable form for ease of traceability.

The lifetime of the generated node is controlled by the policy specified in the dynamic field in the Target of the local Linkage of the receiving Link Endpoint. Definitionally, the lifetime will never be less than the lifetime of the link which caused its creation.

capabilities

the extension capabilities the sender supports/desires

5.5 std-dist-mode: *symbol(Link distribution policy)*

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

5.6 filter (the predicate to filter the Messages admitted onto the Link)

```
<type class="composite" name="filter" source="list">
```



```
<descriptor name="amqp:filter:list" code="0x00000001:0x0000020a"/>
  <field name="filter-type" type="symbol" mandatory="true"/>
  <field name="filter" type="*" mandatory="true"/>
</type>
```

Field Details:

filter-type	the type of the filter
filter	the filter predicate

5.7 filter-set: *map*

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

5.8 delete-on-close (lifetime of dynamic Node scoped to lifetime of link which caused creation)

```
<type class="composite" name="delete-on-close" source="map" provides="lifetime-policy">
  <descriptor name="amqp:delete-on-close:map" code="0x00000001:0x0000020b"/>
  <field name="options" type="options"/>
</type>
```

Field Details:

options	options map
---------	-------------

5.9 delete-on-no-links (lifetime of dynamic Node scoped to existence of links to the Node)

```
<type class="composite" name="delete-on-no-links" source="map" provides="lifetime-policy">
  <descriptor name="amqp:delete-on-no-links:map" code="0x00000001:0x0000020c"/>
  <field name="options" type="options"/>
</type>
```

Field Details:

options	options map
---------	-------------

5.10 delete-on-no-messages (lifetime of dynamic Node scoped to existence of messages on the Node)

```
<type class="composite" name="delete-on-no-messages" source="map" provides="lifetime-policy">
  <descriptor name="amqp:delete-on-no-messages:map" code="0x00000001:0x0000020d"/>
  <field name="options" type="options"/>
</type>
```

Field Details:

options	options map
---------	-------------

5.11 delete-on-no-links-or-messages (lifetime of Node scoped to existence of messages on or links to the Node)

```
<type class="composite" name="delete-on-no-links-or-messages" source="map" provides="lifetime-policy">
  <descriptor name="amqp:delete-on-no-links-or-messages:map" code="0x00000001:0x0000020e"/>
  <field name="options" type="options"/>
</type>
```

Field Details:

options

options map

Book V - Transactions

1 Transactions

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

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

The container acting as the transactional resource defines a special target that functions as a coordinator. The *transaction controller* establishes a link to this target and allocates a container scoped transaction id (txn-id) by sending a message whose body consists of the `declare` type. This txn-id is then used by the controller to associate work with the transaction. The controller will subsequently end the transaction by sending a `discharge` message to the coordinator. Should the coordinator fail to perform the `declare` or `discharge` as specified in the message, the message will be rejected with a `transaction-errors` indicating the failure condition that occurred. **TODO: ADD DIAGRAM**

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

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

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

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

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

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

TODO: ADD DIAGRAM

The transfer, disposition, and flow frames may travel in either direction, i.e. both from the controller to the resource and from the resource to the controller. When these frames travel from the controller to the resource, any embedded txn-ids are requesting that the resource assigns transactional work to the indicated transaction. When traveling in the other direction, from

resource to controller, the transfer and disposition frames indicate work performed, and the *txn-ids* included MUST correctly indicate with which (if any) transaction this work is associated. In the case of the flow frame traveling from resource to controller, the *txn-id* does not indicate work that has been performed, but indicates with which transaction future transfers from that link will be performed. **TODO: ADD DIAGRAM**

1.1 Transactional Acquisition

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

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

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

2 Coordination

2.1 coordinator (target for communicating with a transaction coordinator)

```
<type class="composite" name="coordinator" source="map" provides="target">
  <descriptor name="amqp:coordinator:map" code="0x00000001:0x00000300"/>
  <field name="capabilities" type="symbol" requires="txn-capabilities" multiple="true"/>
</type>
```

Field Details:

2.2 declare (message body for declaring a transaction id)

```
<type class="composite" name="declare" source="list">
  <descriptor name="amqp:declare:list" code="0x00000001:0x00000301"/>
  <field name="options" type="options"/>
  <field name="global-id" type="*" requires="global-tx-id"/>
</type>
```

Field Details:

options	options map
global-id	global transaction id

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

2.3 discharge (message body for discharging a transaction)

```
<type class="composite" name="discharge" source="list">
  <descriptor name="amqp:discharge:list" code="0x00000001:0x00000302"/>
  <field name="options" type="options"/>
  <field name="fail" type="boolean"/>
</type>
```

Field Details:

options	options map
fail	indicates the transaction should be rolled back

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

2.4 txn-capabilities: *symbol(symbols indicating (desired/available) capabilities of a transaction coordinator)*

Valid values:

- amqp:local-transactions Support local transactions.
- amqp:distributed-transactions Support AMQP Distributed Transactions.
- amqp:promotable-transactions Support AMQP Promotable Transactions.
- amqp:multi-txns-per-ssn Support multiple active transactions on a single session.
- amqp:multi-ssns-per-txn Support transactions whose txn-id is used across sessions on one connection.
- amqp:multi-conns-per-txn Support transactions whose txn-id is used across different connections.

2.5 transaction-errors: *symbol(symbols used to indicate transaction errors)*

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 VI - Security

1 Security Layers

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

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

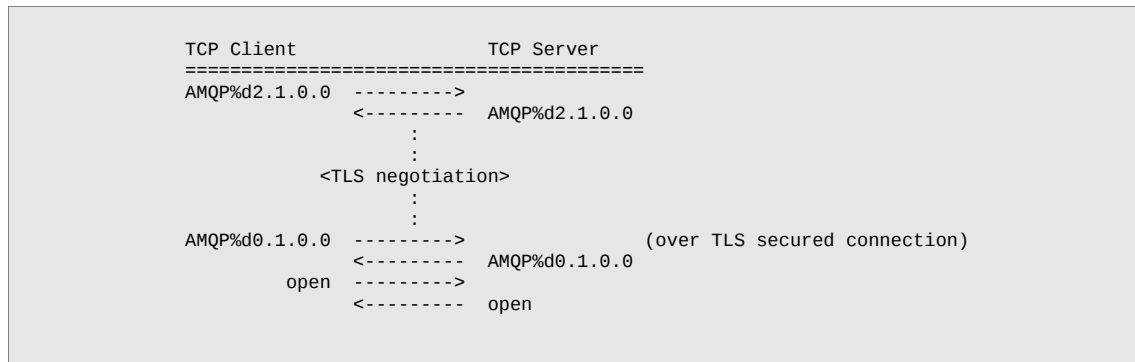
2 TLS

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

4 OCTETS	1 OCTET	1 OCTET	1 OCTET	1 OCTET
"AMQP"	%d2	major	minor	revision

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

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



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

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

3 SASL

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

4 OCTETS	1 OCTET	1 OCTET	1 OCTET	1 OCTET
"AMQP"	%d3	major	minor	revision

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

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

TCP Client		TCP Server
=====		=====
AMQP%d3.1.0.0	----->	
	<-----	AMQP%d3.1.0.0
	:	
	<SASL negotiation>	
	:	
AMQP%d0.1.0.0	----->	(over SASL secured connection)
	<-----	AMQP%d0.1.0.0
open	----->	
	<-----	open

3.1 SASL Negotiation

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

SASL Exchange

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

The peer playing the role of the SASL Client and the peer playing the role of the SASL server MUST correspond to the TCP client and server respectively.

3.2 Security Frame Bodies

3.3 sasl-mechanisms (advertise available sasl mechanisms)

```
<type class="composite" name="sasl-mechanisms" source="list">
  <descriptor name="amqp:sasl-mechanisms:list" code="0x00000001:0x00000400"/>
  <field name="options" type="map"/>
  <field name="sasl-server-mechanisms" type="string" multiple="true"/>
</type>
```

Field Details:

options	options map
sasl-server-mechanisms	supported sasl mechanisms

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

3.4 sasl-init (initiate sasl exchange)

```
<type class="composite" name="sasl-init" source="list">
  <descriptor name="amqp:sasl-init:list" code="0x00000001:0x00000401"/>
  <field name="options" type="map"/>
  <field name="mechanism" type="string" mandatory="true"/>
  <field name="initial-response" type="binary"/>
</type>
```

Field Details:

options	options map
mechanism	selected security mechanism

The name of the SASL mechanism used for the SASL exchange. If the selected mechanism is not supported by the receiving peer, it MUST close the Connection with the authentication-failure close-code. Each peer MUST authenticate using the

highest-level security profile it can handle from the list provided by the partner.

initial-response

security response data

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

3.5 sasl-challenge (security mechanism challenge)

```
<type class="composite" name="sasl-challenge" source="list">
  <descriptor name="amqp:sasl-challenge:list" code="0x00000001:0x00000402"/>
  <field name="options" type="map"/>
  <field name="challenge" type="binary" mandatory="true"/>
</type>
```

Field Details:

options

options map

challenge

security challenge data

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

3.6 sasl-response (security mechanism response)

```
<type class="composite" name="sasl-response" source="list">
  <descriptor name="amqp:sasl-response:list" code="0x00000001:0x00000403"/>
  <field name="options" type="map"/>
  <field name="response" type="binary" mandatory="true"/>
</type>
```

Field Details:

options

options map

response

security response data

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

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

```
<type class="composite" name="sasl-outcome" source="list">
  <descriptor name="amqp:sasl-outcome:list" code="0x00000001:0x00000404"/>
  <field name="options" type="map"/>
  <field name="code" type="sasl-code"/>
  <field name="additional-data" type="binary"/>
</type>
```

Field Details:

options

options map

code

indicates the outcome of the sasl dialog

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

additional-data

additional data as specified in RFC-4422

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

3.8 sasl-code: *ubyte(codes to indicate the outcome of the sasl dialog)*

Valid values:

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