AMQP 1.0 DRAFT for discussion Revision 200 JO/RS/RG

UNPUBLISHED CONFIDENTIAL MATERIAL OF

THE AMQP WORKING GROUP

DISTRIBUTION TO MEMBERS AND REVIEWERS ONLY

Table of Contents

1 Introduction (TODO)....5
1.1 Overview....5
2 Logical Model....6
2.1 Nodes and Links....6
2.2 Messages....7
2.3 Credit....8
2.4 Containers....9
2.5 Sessions....10
2.6 Commands....10
2.7 Transactions....12
2.7.1 Local Transactional Mode (TODO)....12
2.7.2 Distributed Transactional Mode (TODO)....12
2.8 Requirements for a Transport (TODO)....12
3 Data Types....14
3.1 Primitive Types....14
3.1.1 Integral types:....14
3.1.2 Floating point and decimals....14
3.1.3 Other Primitive Types....14
3.1.4 Compound Types....15
3.2 Programming Language Mappings....15
4 Links....16
4.1 Link Properties....16
4.2 Transferring Messages....17
4.3 Message Receipt....17
4.4 Link Modes....18
4.4.1 Destructive Link....18
4.4.2 Non-Destructive Link....18
4.4.3 Link Mode Uses....19
5 Messages....21
5.1 Description....21
5.1 Message Properties....21
5.2 Message Identity....22
5.3 Property Names....22
5.4 Standard Properties....22
5.5 Standard Message Encoding....23
6 AMQP Message Brokers....24

6.1 Overview....24
6.2 Queues....24
6.2.1 Purpose....24
6.2.2 Description....24
6.2.3 Queue Properties....25
6.2.4 Common Queue Configurations....26
6.3 Services....28
6.3.1 Broker Management (amqp$admin)....28
6.3.1.1 Command Encoding (TODO)....29
6.3.1.2 Responses Results (TODO)....29
6.3.1.3 Error Handling (TODO)....29
6.3.2 Inter-Broker Transfer (amqp$transfer)....30
6.3.2.1 Addressing....30
6.3.2.2 Aliasing of Transfer Service....30
6.3.3 Distributed Transactions (amqp$dtxCoordinator) (TODO)....32
7 Using AMQP....33
7.1 Single Broker Topologies....33
7.1.1 Point-to-Point: Single Producer/Consumer....33
7.1.2 Point-to-Point: Shared Work Queue....33
7.1.3 Point-to-Point: Shared Work Queue Using Filters....34
7.1.4 Publish/Subscribe: Transient Pub/Sub....35
7.1.5 Publish/Subscribe: Durable Pub/Sub (TODO)....36
7.2 Multiple Broker Topologies....36
8 Glossary....37
8.1 Message (TODO)....37
8.2 Container....37
8.3 Node....37
8.4 Address....37
8.5 Link....37
8.6 Session (TODO)....38
8.7 Queue (TODO)....38
8.8 Broker....38
8.9 Client....38
8.10 Transaction (TODO)....38

1 Introduction (TODO)

1.1 Overview

AMQP is a standard for building a message based networks. AMQP is defined in layers which build upon each other. At the lowest layer it defines an efficient binary peer-to-peer message transfer wire protocol. On top of the wire protocol sits a powerful logical model of nodes and links, used to establish a global routing and addressing network. Using the logical model we define a concrete set of queuing semantics which must be respected by a compliant AMQP Message Broker.

AMQP defines a standardized wire-level protocol, a common model for broker semantics across point to point and publish/subscribe messaging, and a global addressing scheme to support the sending of messages between organisations.

The standard provides definitions in three areas: the overall architecture of an AMQP Network, the architecture of a Broker serving as a store and forward node within an AMQP Network, and the peer to peer protocol used between AMQP Nodes.

AMQP defines a standard for Message Brokers. It is in the Application Layer of the 4 layer TCP/IP stack. AMQP provides a stateful communications facility above the stateless peer-to-peer networking capabilities of TCP/IP.

AMQP specifies discrete system components which together form a Message Broker (Broker). The Broker provides secure, asynchronous, interoperable store-and-forward message queuing, publish/subscribe messaging and delivery services for Client applications.

AMQP Client applications arrange to pass data among themselves using a Broker as a trusted intermediary and Queues as named rendezvous therein. By doing so the Client applications can communicate even though they may execute at different times and in different places and within different administrative domains.

AMQP implements an overlay network on top of the Internet. The AMQP overlay network is a graph where the nodes are reliable (often persistent) message stores and the edges are secure links which move messages between the stores. A store can be a queue, a broker or a client. Links can connect clients to brokers, connect queues to other queues, or connect brokers to brokers across the Internet.

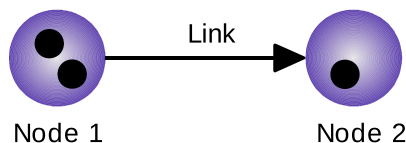
2 Logical Model

2.1 Nodes and Links

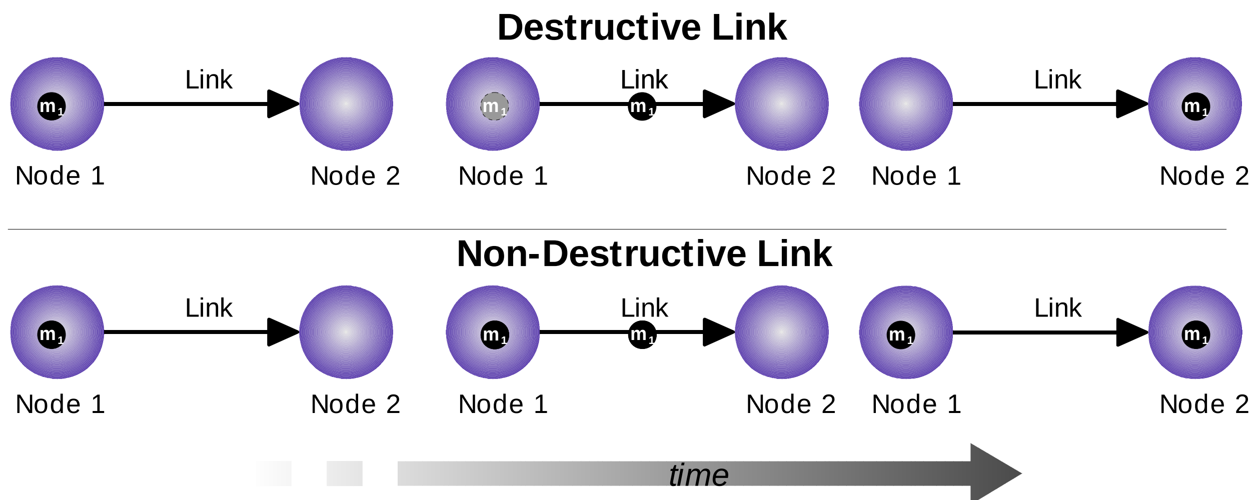
An AMQP Network consists of **Nodes** and **Links**.

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

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



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

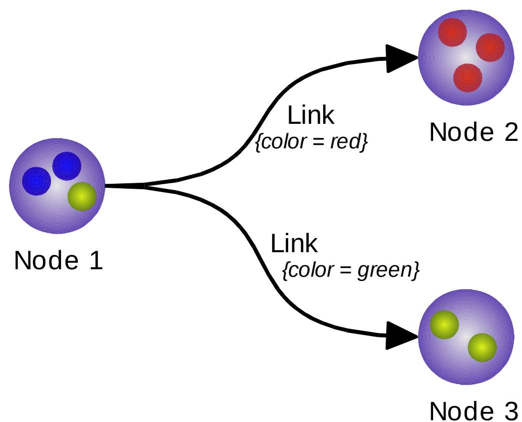


In the Non-Destructive Link example the message m_1 is now simultaneously present at more than one node. In AMQP a message can be in many places at once. More correctly one can think of the single message being referenced from many different places. Logically the AMQP network could consist of a routing network which transfers only the message identity, and a single globally accessible repository from which the message properties and bodies can be retrieved using the message identity as the key.

A message is either present at a given node, or it is not. Messages may not be present multiple times. That is, if a message *M* arrives at a node *N* where *M* is already present, then there is no change to the set of messages at *N*. More generally nodes should treat as a duplicate the arrival of any message *M′* which has the same identity as a message a *M* which has previously arrived at the node. Duplicate deliveries should be discarded.

2.2 Messages

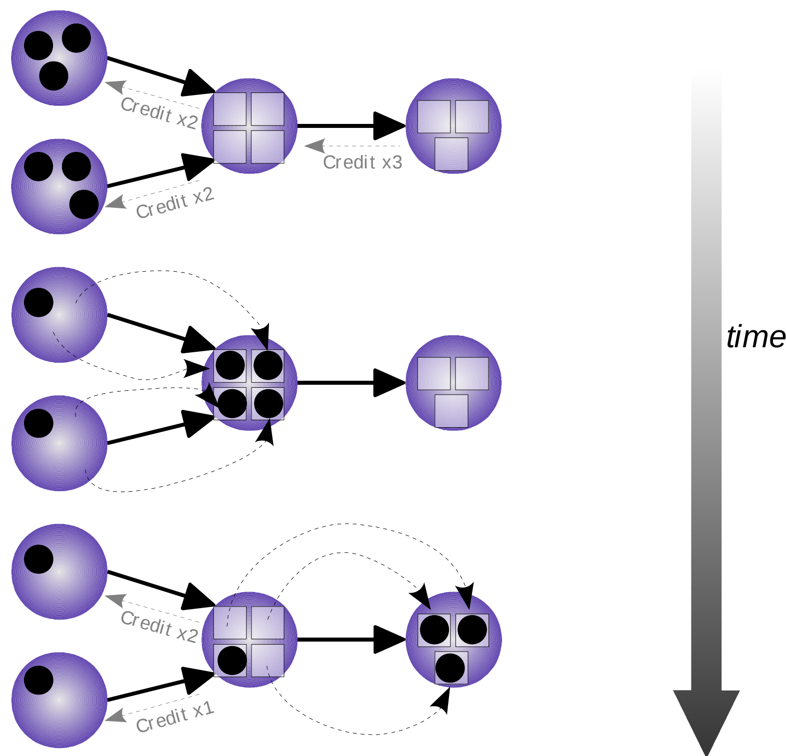
A **message** is a uniquely identified unit of application data consisting of readable “properties” and opaque “body”. By using filters on the outgoing links from a node, messages may be distributed based on the value of message properties.



2.3 Credit

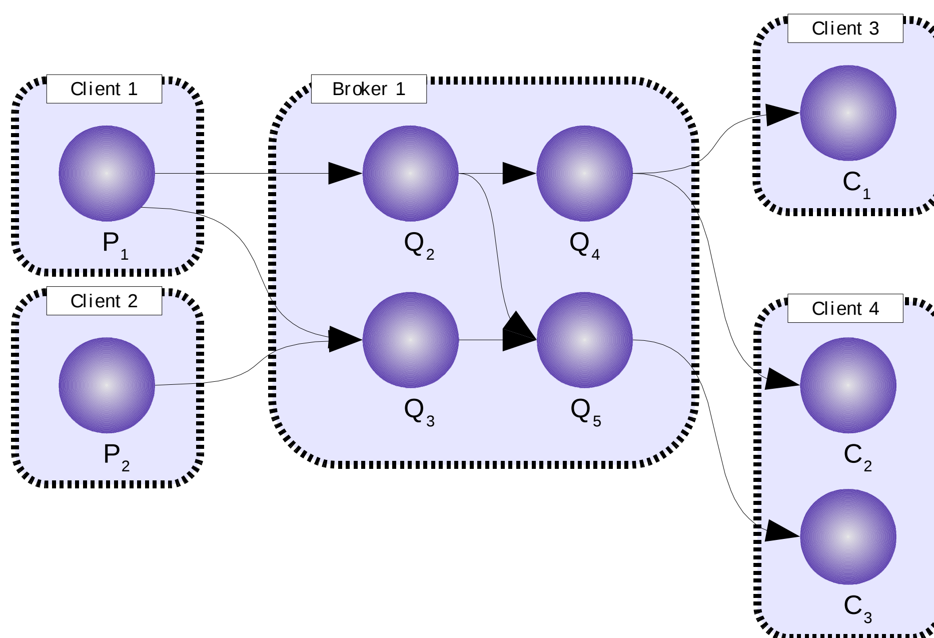
A message may only pass along a given link if the destination node has issued **credit** to the link. A link with no credit is essentially inactive. One unit of credit allows one message to be passed along the link. In order for a second message to be transferred, a second unit of credit needs to be issued. Destination nodes can add or remove credit at any time.

One can think of AMQP as a network of messages travelling in one direction, and an equal (or greater) amount of credit flowing in the opposite direction. Credit is used to ensure that nodes to not receive more messages than they can physically store. If we think of each node having a fixed capacity then we can see that such a node can distribute an amount of credit equal to the free-space at that node amongst its incoming links.



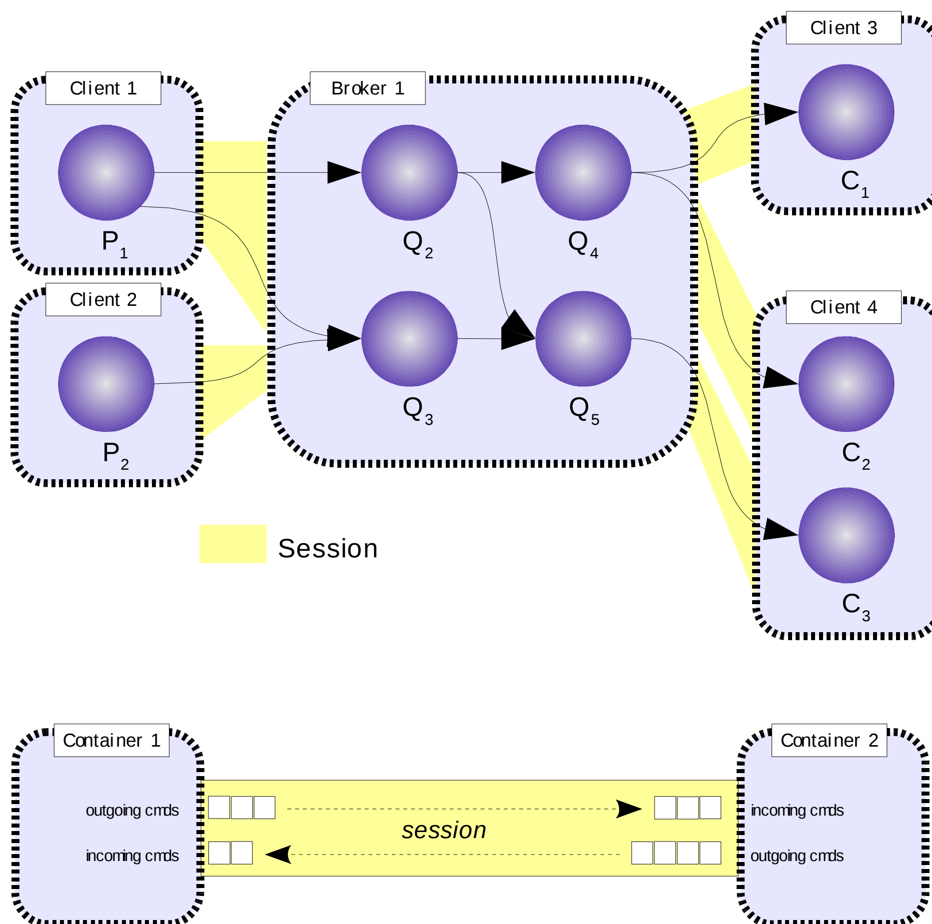
2.4 Containers

Nodes within the AMQP network exist within **Containers**. A container is a physical or logical process to which network connections can be established. Node names are unambiguous within a container (that is no two nodes will simultaneously have the same name within the container, though a node may simultaneously more than one name). Containers have a globally unique identifier. The combination of container identifier and node name is thus a globally unambiguous identifier for any node. While container identifiers are unique they are not user-friendly. More commonly a container will be referenced by its network address or (where appropriate) DNS name and port. AMQP defines the behaviour of a specific type of container: the AMQP Message Broker.



2.5 Sessions

A link may be between two nodes within the same container, or between nodes in different containers, the former being just a special case of the latter. Links between different containers must be created on a **Session**. A session is a named interaction between two containers providing for a pair of reliable ordered stream of commands (one in each direction). Establishing a session requires (mutual) authentication. There may be more than one session established between any two containers at any one time. Each session may contain multiple links.



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

2.6 Commands

Sessions are a transport for commands. Commands are the atomic units of work of the AMQP transport protocol. Commands are used to create links between nodes in the source and destination containers, to transfer message data, and to issue and revoke credit. In general an AMQP session will be carried over some form of network layer, thus commands sent on a session are asynchronous. In order to be sure that a

particular action has been executed by the receiving container, the sending container must wait for confirmation of completion. AMQP does not allow for (observable) out-of-order execution.

In order to maximize throughput any number of new commands may be sent before confirmation of completion of the previous command. Should a command fail for some reason, then the sender of the command will be notified as to the last successfully executed command. No command after the failed command will be executed.

2.7 Transactions

Applications may require that message groups of related message transfers be actioned together in a single atomic transaction. AMQP provides three transactional models:

1. Non transactional mode
2. Local transactional mode
3. Distributed transactional mode

In non-transaction mode each message transfer forms an atomic action. For Local and Distributed transactional modes the units of work (transactions) have to be demarcated in the command stream.

The transaction mode is controlled at the session level, is set at the time of session creation and cannot thereafter be modified. Two sessions between the same two containers do not need to have the same transactional mode. Session level transactions allows for the common case of wanting to combine into a transaction the actions of a message arriving via one link, work being performed, and a response being sent over a second link.

In modes other than non-transactional on container takes the role of the Transaction Controller, the other as the Transactional Resource. These roles are fixed at the time of session creation. The container acting as the Transaction Controller demarcates the transactions in the command stream.

Only state changes related to message transfers are affected by transactions, other state changes (such as the creation or deletion of links, the issuance of credit, etc) are not.

2.7.1 Local Transactional Mode (TODO)

In local transactional mode, when the container acting as the Transaction Controller for the session marks the end of the current transaction it informs the Transaction Resource as to the last command in the incoming stream. That lets the resource know exactly which commands (in both directions) are considered to be part of the transaction.

In local transactional mode, messages transfered from the controller to target nodes on the resource are not made visible to outgoing links from the target node until the transaction is committed. Messages transferred from source nodes on the resource to target nodes on the controller are not archived as a result of successful acceptance until the transaction they are part of is committed.

When a local transaction is rolled back, uncommitted transfers from sources on the controller to targets on the resource are removed; and uncommitted acceptance of transfers from the resource to the controller are discarded.

2.7.2 Distributed Transactional Mode (TODO)

2.8 Requirements for a Transport (TODO)

A Transport for AMQP must provide the necessary session interface to a container. That is it must be able to

- Establish links between a local and a remote nodes in each direction
- Transfer a message (or arbitrary size) along such a link along with support header and footer data
- Issue and revoke credit along incoming links
- Accept, release or reject transferred messages
- (Optionally) request a transferred message to be parked
- (Optionally) request a parked message to be accepted, rejected or released
- Demarcate transactional boundaries

3 Data Types

AMQP specifies a logical type system. Encoding of the types is specified in the AMQP Message Format and AMQP Transport.

3.1 Primitive Types

3.1.1 Integral types:

Type	Minimum Value	Maximum Value	Description
byte	-128	127	
short	-32768	32767	
int	-2147483648	2147483647	
long	-9223372036854775808	9223372036854775807	
unsignedByte	0	255	
unsignedShort	0	65535	
unsignedInt	0	4294967295	
unsignedLong	0	18446744073709551615	

3.1.2 Floating point and decimals

Type	Description
float	32-bit floating point type (IEEE float)
double	64-bit floating point type (IEEE double)
decimal	Decimal numbers with up to 16 digits (IEEE decimal64)
longDecimal	Decimal numbers with up to 34 digits (IEEE decimal128)

3.1.3 Other Primitive Types

Type	Description
character	Single unicode character
string	Text string of unicode characters (up to 2^{28} - 1 unicode characters long).
dateTime	Specific instant of time, with granularity up to milliseconds (RFC3339).
binary	A sequence of binary octet data of up to 2^{32} - 1 octets in length.
boolean	A Boolean true or false value.

3.1.4 Compound Types

Type	Description
map	
array<*type*>	

3.2 Programming Language Mappings

Normative Representations

```
|AMQP    |  SQL92   XDM        C99       Java    .NET
|---------------------------------------------------------
|boolean |  boolean xs:boolean bool      boolean boolean
|byte    |  ?       xs:byte    char      byte    byte
|short   |  short   xs:short   short     short   short
|int     |  int     xs:int     int       int     int
|long    |  long    xs:long    long      long    long
|float   |  float   xs:float   float     float   float
|double  |  double  xs:double  double    double  double
|string  |  String  xs:string  char[](*) String  String
```

4 Links

4.1 Link Properties

Links have properties which control their behaviour. The following Link properties are specified by AMQP but an implementation may have additional configuration options.

name: *string*

Each link named. The name is unique with respect to the container in which the link resides. For links whose lifetime is scoped to that of the session which created them the link name must begin with the session identifier. For non-session scoped links which bridge between containers the name must begin with the container-id of the remote container.

temporary: *boolean*

A temporary link has a lifetime which is tied to the session on which it was created. Such a link will automatically be destroyed when the session which created it is destroyed. A non-temporary link should be treated as durable – it should survive as long as other durable data held by the container survives. A non-temporary link will be removed only when either the source, or destination node is destroyed, or when the link itself is explicitly destroyed. Once created a link may not change from temporary to non-temporary or vice versa.

mode: **destructive | non-destructive**

The link mode, see Section 4.4 Link Modes below. The link mode may not be altered after the creation of the link.

filter-type: *string*

Where a filter has been supplied for the link, the filter-type defines how the query language used for the filtering. AMQP Message Brokers (see Chapter 6) must support the "SQL-92" filter-type. If no filter is used on the link then this field may be left empty.

filter: *string*

The filter to be applied as an entry criteria for messages attempting to pass along this link. The interpretation of this field depends on the fitler-type specified. If this field is left empty then no filter will be applied.

first-message-date: *dateTime*

The first-message-date property may be set on creation of the link, but may not be altered thereafter. If set then on creation of the link only message which arrived at the source not at or after the specified dateTime will be sent along the link.

If the dateTime specified is in the future it will be adjusted to the current time. Thus if the link creator wishes only for messages which arrive at the source node to be sent to the target, and not historical messages, then setting the first-message-date to the maximal value for dateTime will provide the desired behaviour. This may be the desired outcome if the node link is acting as a topic-subscription.

Setting the first-message-date to the minimal value for dateTime ensures that all available historic messages will be considered for sending along the newly established link. This may be the desired outcome if the link is acting as a consumer on a queue.

4.2 Transferring Messages

When a message is transferred along a link it is accompanied by supporting data about the message. Some data associated with a message changes over time, or relates to and is altered by the passage of the message through the AMQP Network. Such data cannot be part of the message proper as AMQP guarantees not to alter the message through its journey. Furthermore the message may be in many places at once in the network, and this supporting data may not be the same in all places.

This supporting data takes the form of "headers" and "footers" transferred alongside the message across the link between nodes.

Defined headers are

priority: *byte*

The relative message priority (where 0 is the lowest priority).

durable: *boolean*

Durable messages must not be lost even if an intermediary container is unexpectedly terminated and restarted. The implication is that durable messages must be stored on some reliable media.

non-transacted: *boolean*

Non-transacted messages do not participate in transactions. A non-transacted message sent to a queue on the transaction resource will be enqueued whether an explicit transaction-commit is issued or not.

expiration: *dateTime*

The time at which the message may be considered "expired". Expired messages can go straight to the "archived" state from the "available" state on a node.

TODO: Security headers

TODO: Security footers (signatures)

TODO: Audit Trail Footers

4.3 Message Receipt

On receipt of a message, the target node may take one of the following actions

- accept the message indicating a successful transfer of responsibility
- release the message indicating that the target node did not want to accept the message at this time, but may do so if the message is resent in the future
- reject the message indicating the nature of the failure conditions
- park the message keeping the message available to the link for acceptance (or release) at a later time.

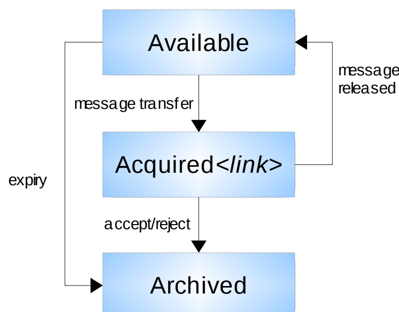
Each message must be processed in order, although "parking" a message allows the receiver to defer the final disposition. Parking is the optional ability for a node to defer acceptance of a message until a time of its choosing, even from within a transaction. Messages which are parked are still associated with the Link which was used to Park them. When a link closed, any parked messages on that link are released back to the source node. Parking is scoped to the link and not to a transaction. If a message is parked on a transaction-enabled session, rolling back the transaction does not release the message – it remains parked.

4.4 Link Modes

4.4.1 Destructive Link

A destructive link moves messages from its source node to its target node. A message sent along a destructive link is not available to be sent along any other link unless and until the destructive link releases the message.

Destructive links update the state of the message at the node (as opposed to non-destructive links, which do not – see below). Before a message is transferred along a destructive link the message is first "acquired" by the link. Acquiring the link makes it unavailable to other links. A message in the "acquired" state at the node can either become "archived" if the target node accepts or rejects the message, or the message may be released back to the source node to become "available" once more.



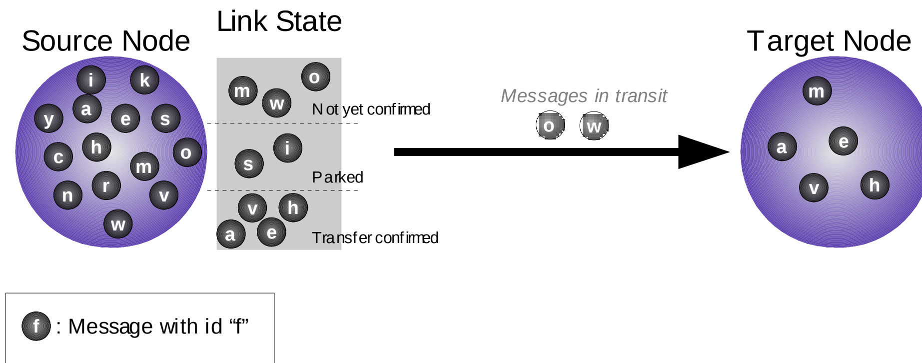
Messages in the "acquired" state at the node need to track which link has acquired them. Within the acquired state there is further state regarding the message which is held at the link end (rather than at the node). The link end holds a record of all messages which the link currently has acquired, and whether each of those messages is currently "parked" on the link or not.

4.4.2 Non-Destructive Link

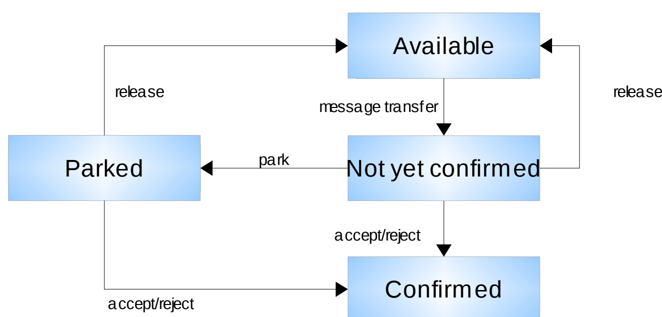
A non-destructive link copies messages from the source node to the destination node leaving the original reference at the source node unchanged.

Non-destructive links track the state of messages transferred along them. At the source end of the link a record needs to be kept of which messages have been confirmed transferred (either accepted or rejected), which are currently parked, and which have been transferred but for which no confirmation has yet been received. Only messages which are in the "available" state (see above) on the node are eligible to be offered to a non-destructive link. Between the time that the message is transferred on the non-destructive link and

the time it is confirmed by the target node the message may change state due to the actions of destructive links.



To avoid repeatedly delivering the same message again and again, the link will check each node it is offered by the source node against the records of messages it has in the "not yet confirmed", "parked" and "confirmed" states. Thus we get a state diagram as follows:



Note that this state is tracked by the (non-destructive) link and, and not by the node. The same message may be in many different states on the same node but on different non destructive links. While the message may return to the "available" state on the link it may no longer be available on the node due to the actions of destructive links. In this case the message cannot be re-offered to the non-destructive link unless it becomes available again.

Once the target node of the link has irrevocably confirmed the successful receipt of a message transferred over a non-destructive link, the source node is made aware that the message need no longer be offered to the given link.

4.4.3 Link Mode Uses

Destructive links are in general used when building point-to-point messaging systems and "work sharing" queues. Destructive links ensure that each message is only delivered (successfully) once from each node. It is expected that nodes with multiple destructive links distribute available messages with an approximation to fairness amongst the outgoing links capable of handling them.

Non-Destructive links can be used in conjunction with Destructive links on the same node. N this case non-destructive links provide a “browsing” function, allowing the target of the link to view available messages at the source node without removing them.

Where permits only non-destructive outgoing links it can be used to provide publish/subscribe topic-like behaviour. Each link sees a copy of the messages which come in (filtering out those which it is not interested in). The links can be created with options such that at the time of joining the source node it will only receive messages which arrive after the link has been established. Where a node is acting in this manner it must implement some policy to move the state of messages at the node from “available” to “archived”. Two possible policies are to use time based expiry, or to archive messages when all connected links have examined the message.

5 Messages

5.1 Description

An AMQP Message is a sequence of octets that is guaranteed to be transported unaltered through the AMQP network. Messages are uniquely identified by the originator using a “message-id”. Clients transfer Messages to Queues with the intent that other Clients retrieve them at some later time (which may vary between microseconds to days depending on the Client application). A single message may transfer through many queues and be delivered to many consumers.

Logically, an AMQP message consists of two parts

- **Body** - The Body is the main content of the Message. The Broker is not required to interpret the contents of Message Body (though vendor extensions may do so).
- **Properties** - Contain additional information relating to the Message which is visible to the Broker; the Broker may interpret the Properties and apply filtering and selection predicates to the Message based on the Properties. AMQP defines standard Properties which affect broker and client behaviour on processing the message. Other Properties may be defined by applications.

AMQP places no restrictions on the maximum size of messages. AMQP defines both a logical and a physical message format. The logical format defines how Properties may be addressed from within a broker, or client. By separating the logical interface from the physical encoding we allow for AMQP to transport messages in legacy formats. For example, for compatibility reasons separate physical formats are used to allow the transportation of messages generated in pre 1-0 versions of AMQP.

5.1 Message Properties

Message Properties are named type-value pairs which can be retrieved by processing the message. Since Properties are part of the message, once the message has entered the AMQP network, the values associated with the Properties cannot change as this would invalidate the immutability contract of AMQP. Logically the Properties can be thought of as a map of Name to Typed-Value.

There are two types of Property:

- Standard Properties defined in the AMQP specification and the values of which may change the behaviour of brokers or clients processing the message.
- Application Properties which are set by the sending Client..

Both types of Properties may be addressed in Filters,

It is possible for a Client to encode their application data using the Properties alone. Property information is intended to be smaller than Body information.

Normatively, the Client gains access to Properties using its Name to access its Value of its specified Type. Access must be by Name in order to support the application of SQL-92 predicate logic to properties during Message routing and delivery.

5.2 Message Identity

A Message is a uniquely identified unit of data in the Message Queue Model.

The Message is identified universally by its `amqp.message-id` property, which is a Standard Property. This identity is intended to be globally unique, and is set by the originator of the message.

If a Node sees two Messages which share the same `amqp.message-id` then it considers Messages to be identical regardless of other content and may discard the later Message.

5.3 Property Names

Properties names are namespaced within separate domains to prevent collisions between standard AMQP Properties, vendor extensions, and application specific Properties.

Standard Property names all are within the "amqp" namespace and may be addressed as `amqp.`<*property name*> - e.g. `amqp.message-id.` All Properties within the amqp namespace are reserved for use by the AMQP speicifcation. It is an error to attempt to set a Property in the amqp namespace if the Property is not defined within the specification.

Vendors of AMQP intermediaries wishing to add extensions to the standard Properties should use the domain

`amqpx.`<*subdomain*>

Where *subdomain* is an unambiguous vendor name, product, name or open-source project name or similar.

Application specific Properties must be placed in the `application` domain, for instance `application.color` would be a valid Property name.

When referencing Property names from filters, the domain may be ommitted. In this case the container in which the filter is being applied with infer a domain: if a standard Property exists with the supplied name it will infer that is the Property desired, else if a Property with the supplied name exists amongst the supported vendor extensions then that will be used, else it will assume an application Property is meant. Care should be taken when omitting domain names, as future versions of the AMQP specification may contain new Properties which lead to collisions and thus the semantics of existing link filters changing.

5.4 Standard Properties

The following standard Properties are defined by AMQP

message-id: *binary* — application message identifier

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

user-id: *binary* — creating user id **(optional)**

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

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

This property identifies the node that is the intended destination of the message, it is in the form of an AMQP Address.

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

The AMQP Address of the node to send replies to.

correlation-id: *binary* — application correlation identifier **(optional)**

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

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

The total size in octets of the opaque message body.

content-type: *string* — MIME content type **(optional)**

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

5.5 Standard Message Encoding

The AMQP Transport allows for alternative encoding of messages as long as a mapping to the logical model can be performed. The Standard Encoding is to be used except where the AMQP network is being used to transport messages of legacy protocols (in particular pre 1-0 versions of AMQP).

6 AMQP Message Brokers

6.1 Overview

An AMQP Message Broker (a Broker) is a Container with well defined semantics. Brokers are containers for nodes known as Advanced Message Queues (Queues). Brokers also provide nodes which correspond to well defined services: in particular a service exists for managing the broker as well as the nodes and links contained within it, and a service exists for managing the distribution of messages into a network of Brokers using a well defined global addressing scheme.

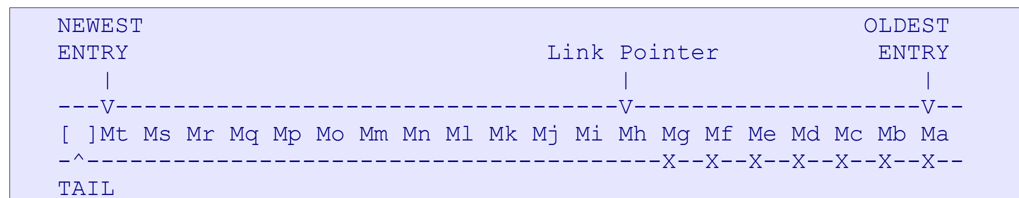
6.2 Queues

6.2.1 Purpose

An Advanced Message Queue (Queue) is a node that holds Messages for eventual distribution through outgoing links. Queues provide (within certain restrictions outlined below) ordering guarantees on messages passing through them. Queues also allow for the provision of reliability guarantees on message delivery.

6.2.2 Description

The following illustration shows the simplest Queue:



```
NEWEST                                                        OLDEST
ENTRY                                 Link Pointer             ENTRY
   |                                       |                     |
---V-----------------------------------V---------------------V--
[ ]Mt Ms Mr Mq Mp Mo Mm Mn Ml Mk Mj Mi Mh Mg Mf Me Md Mc Mb Ma
-^------------------------------------------X--X--X--X--X--X--X--
TAIL
```

The Head of the Queue is the Entry which would be removed by the next dequeue operation. The Tail of the Queue is the Entry most recently enqueued.

A Queue may have more than one Head – each Head being represented and managed by a Link which has the Queue as its Source.

Queues provide a limited FIFO guarantee. For Entries of equal Priority on the Queue, delivery along a given Link will always be attempted in the order the Entries were placed on the Queue. If Entries have different Priorities, then higher Priority items may be submitted to the Link first.

The first attempt to transfer a Queue Entry (of a given Priority) to a Link will be made in order of arrival into the Queue. If the first attempt fails due to explicit or implicit release of acquired entries or transaction rollback, then subsequent attempts may result in messages being transfered out of order on a given link. For example, if Sessions A and B both have Links to Queue Q, and Message M is delivered to Session A from Queue Q, but then subsequently releases that Message, then Message M may then be delivered to Session B even though Session B has already been sent Messages which are more recent than M – thus to Session B the Message M is delivered out of order.

Clients can freely create, share, use, and destroy Queues and Links, within the limits of their authority. Alternatively Queues and Links may be externally configured by a Broker administrator, which is common practice.

Queues hold their messages in memory, on disk, or some combination of these. All compliant implementations MUST honour the durability guarantees asserted herein.

Queues maintain the invariant that at any given point no two entries will ever refer to two messages with the same message-id.

6.2.3 Queue Properties

Queues have properties which control their behaviour. The following Queue properties are specified by AMQP but an implementation may have additional configuration options.

name: *string*

The Queue Name is the (the node name) is, as previously defined, unique within the container. Generally, when applications communicate via a Queue they agree on a Queue Name beforehand. A Queue name must not be the empty string, nor must the name be so long that it cannot be represented in less than 2Gb of unicode (in practice transports will provide a much lower limit on the maximum usable queue name length). The FIRST character must be limited to letters a-z or A-Z, digits 0-9, or the underscore character ('_'); all other characters can be any be legal unicode character. Names beginning `amqp$` are reserved for amqp services.

temporary: *boolean* (Default: **false**)

A temporary queue has a lifetime which is tied to the link for which it was created. Such a queue will automatically be destroyed when the link which created it is destroyed (and if that link is itself temporary this will be no later than when the session on which it was created is destroyed). A non-temporary queue should be treated as durable – it should survive as long as other durable data held by the container survives. A non-temporary queue will be removed only when the queue is explicitly destroyed. Once created a queue may not change from temporary to non-temporary or vice versa.

capacity: *int*

The maximum number of messages the queue may hold. Message at the queue in the “available” or “acquired” states are considered to be held by the queue. Messages that are archived do not count towards the capacity restriction.

The capacity of the queue may be used by the broker to determine the amount fo credit to be distributed to incoming links.

message-durability-mode: **message | force-durable | force-transient** (Default: **message**)

Determines the treatment of durable/non-durable messages if they are enqueued on the queue. Durability of messages is generally a property of the message, however it makes no sense to store durably data which is only referenced by queues which are not themselves durably stored. Further, some queues may wish to always store data related to messages enqueued on them regardless of the request of the originator of the message.
Setting this property to **message** simply means that the queue will respect the durability

property of the message. Setting this property to **force-durable** ensures that for as long as the message is enqueued on the queue it will be durably stored by the Broker. Setting this property to **force-transient** means that on restarting the broker all entries that were present in the queue will no longer be found in the queue (the same message may, however, be stored durably by other queues).
A temporary queue which was created by a temporary link should always use **force-transient**.

non-transactional-support: **all** | **unsupported** | **only** (Default: **all**)

Defines whether the queue supports "non-transactional" messages (as defined by the non-transacted" message header). A setting of **all** indicates the queue supports normal and "non-transacted" messages. If set to **unsupported** then non-transactional messages are not supported and messages with non-transacted set to true will be rejected. A queue with non-transactional-support set to **only** will only accept non-transactional messages. This setting is normally only used for system services where the use of transactions may cause deadlocks.

maximum-destructive-links: *int*

The maximal number of destructive links that can be actively using this queue as their source node. If not set, the number of links is unbounded (or more correctly limited only by system resources and policies). If set to 0 then only non-destructive links are allowed.

excess-destructive-link-policy: **fail** | **wait** (Default: **fail**)

Where the maximum-destructive-links property is set to a non-zero value, the queue requires a policy for dealing with requests to create a destructive link which push the number of active links above the specified limit.

If set to **fail** then the attempt to create the new link will return an error to the requester. If set to **wait** then the attempt will appear to succeed, however the link will not receive any messages until some other currently attached destructive link is removed. At that point a **wait**ing link will be chosen to take the place of the link which has been destroyed.

archive-expired: *boolean* (Default: **true**)

If set then messages on the queue are checked to make sure that they have not passed their defined "expiration" (as defined by the transfer headers). Where this property is set and the message is set, then the message is transitioned into the archived state.

archive-currently-unreachable: *boolean* (Default: **false**)

If set then the queue will move a message into the "archived" state if all currently attached links have already seen the message and either irrevocably accepted or rejected it, or have declined interest in it (either due to it failing to meet filter criteria, or because the message is too old). Note that if there are no links currently attached and this property is set, then all messages at the node will be archived. This setting is used to provide publish/subscribe topic like behaviour.

6.2.4 Common Queue Configurations

There are some common Queue lifecycles:

1. **Durable Message Queues** which are Queues shared by Links from many Sessions. Such Queues have an independent existence and continue to collect messages whether or not there are outgoing Links to receive them. Such Queues will usually hold Persistent Entries, but may hold a mixture of Persistent and Non-Persistent Entries. Durable Message Queues are commonly pre-configured by

the Broker administrator. If the Queue is a critical application resource it will often be set to Persistent to ensure safe storage off all the Messages in it..

2. **Temporary Message Queues** are scoped to the creating Link. Where the Link is not durable, the lifetime is thus tied to the lifetime of the Session that created the Link. Such Queues will usually hold only Non-Persistent Entries and are most commonly found as reply queues in the Request/Response use case.

3. **Durable Topic Subscription Queues**; which are Durable Queues with multiple (durable) Links administratively associated with them because they require a well-known name to be of use to the Client application. Such queues provide an efficient way to process the Publish/Subscribe use case where Persistent Entries are required.

6.3 Services

Services are applications within the broker that operate by reading messages sent to special named nodes within the broker.

6.3.1 Broker Management (amqp$admin)

Manipulation of the model entities (Queues and Links) can be carried out using the in-built AMQP Broker Management service.

The Broker Management Service uses the node amqp$admin as the input to its command processor. Management commands are sent as messages to this node. The command processor processes the commands and replies with a results message.

Management commands are sent as a batch of one or more commands within a single AMQP message. The batch of commands is executed atomically, either all commands within the batch complete, or none do. To prevent deadlocks, the amqp$admin service will only accept messages with the non-transacted header set.

Management commands can be queries, assertions, creations, deletions or modifications.

The following commands are supported

- create [or assert] queue <queue name> [with <queue-options>]
- delete queue <queue name>
- assert queue <queue name> [with <queue-options>]
- update queue <queue name> with <queue-options>
- rename queue <queue name> <new queue name>
- query queue [<queue-name>] [with <queue-options>]
- create [or assert] link <link name> <source node> <destination node> [with <link-options>]
- delete link <link name>
- assert link <link-name> [with <link-options>]
- update link <link-name> with <link-options>
- rename link <link-name> <new link-name>
- query link [<link-name>] [with <link-options>]

TODO: define commands for moving/copying messages from queue to queue

6.3.1.1 Command Encoding (TODO)

6.3.1.2 Responses Results (TODO)

The result of executing a management command are returned as a message sent from the amqp$admin service to the reply-to address specified in the properties of the message carrying the original command. The correlation-id property of the response message is set to the message-id of the command message.

TODO: Define response format, error codes

6.3.1.3 Error Handling (TODO)

TODO: Failure of a management command does not cause session failure

6.3.2 Inter-Broker Transfer (amqp$transfer)

An AMQP Message Broker provides a standard service for transferring messages between itself and any other complaint AMQP Broker.

Any message may be sent to the `amqp$transfer` node. The transfer service inspects message sent to this address, and performs routing based on the contents of the "to" message property. The service parses this property as an AMQP address – that is of the form

<node-name>@<broker name>

If no broker name is provided in the address, or if the provided broker name is a known alias for the current broker, then the service forwards the message to the local node with the name <*node-name*>. Otherwise the transfer services uses externally provided configuration to determine the container (or network address thereof) to which is should forward the message for the next stage of it's routing. Logically it then sends the message along a link to the `amqp$transfer` service at that remote broker (in practice intermediate reliable transfer queues may well be used).

6.3.2.1 Addressing

AMQP defines a familiar addressing scheme for communicating between organisations.

For Intranet usage we have:

name@host_name

Where host name is the Fully Qualified Domain Name of the host where the Broker which contains the Queue (or knows how to find it) resides.

For Internet scale deployment we have the familiar:

name@example.com

This has important implications. Internet mail works because DNS contains "MX" (mail exchanger) records that indicate that the mail gateway for "example.com" can really be found at "mail.example.com", for example.

To achieve this effect, AMQP uses a standard DNS SRV (service location) record.

In the example above there would be an DNS SRV record pointing to, amqp.example.com:5672 (it includes the port number) where an AMQP Client (or Broker) can learn how to talk to this server.

This also integrates very neatly with X509 CA checking.

6.3.2.2 Aliasing of Transfer Service

To allow for a more symmetric way of addressing remote nodes, AMQP Brokers automatically alias all node names of the form *x@y* to the transfer service node `amqp$admin`. That is a client may open a link to queue@example.com on the local broker to which they are connected. Such aliases have a responsibility to ensure that message sent to them have the "to" property set to the same address as the node alias name. Special aliases of the form @<broker name> allow the sending of messages to any node at the named remote

broker. Implementations which provide access control may find it easier to block access do amqp$admin and instead grant access only to necessary aliases.

6.3.3 Distributed Transactions (amqp$dtxCoordinator) (TODO)

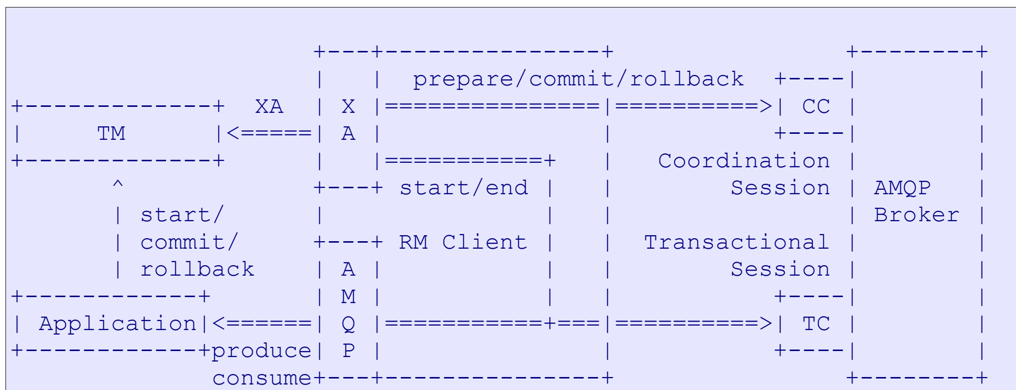
IMPORTANT NOTE:

THIS FACILITY IS SOLELY FOR THE USE OF A TRANSACTION MANAGER TUNNELLING XA COMMANDS TO THE BROKER FOR ONWARD RELAY TO A TRANSACTION CO-ORDINATOR.

DTX IS AN OPTIONAL EXTENSION

The Distributed Transaction provides support for the X-Open XA architecture.

As depicted on the following figure, a Transaction Manager uses the RM Client XA interface to demarcate Transaction boundaries and coordinate Transaction outcomes. RM Clients use the dtx Start and End commands to associate a Transaction with a Terminal. The Transactional Terminal is then exposed to the application driving the Transaction, and may be used to Transactionally produce and consume messages. RM clients use the dtx coordination commands to propagate Transaction outcomes and recovery operations to the AMQP server. A second coordination session can be used for that purpose.

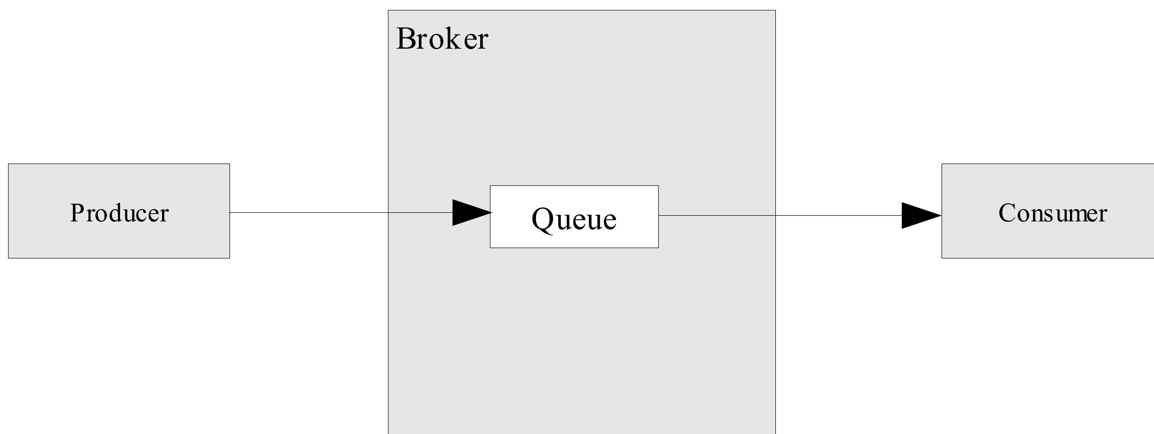


Co-ordination commands are sent to the amqp$dtxCoordinator service.

7 Using AMQP

7.1 Single Broker Topologies

7.1.1 Point-to-Point: Single Producer/Consumer



In the simplest possible case, we have one producer "P", sending messages to one queue "Q" and one consumer "C" which consumes all the messages from "Q".

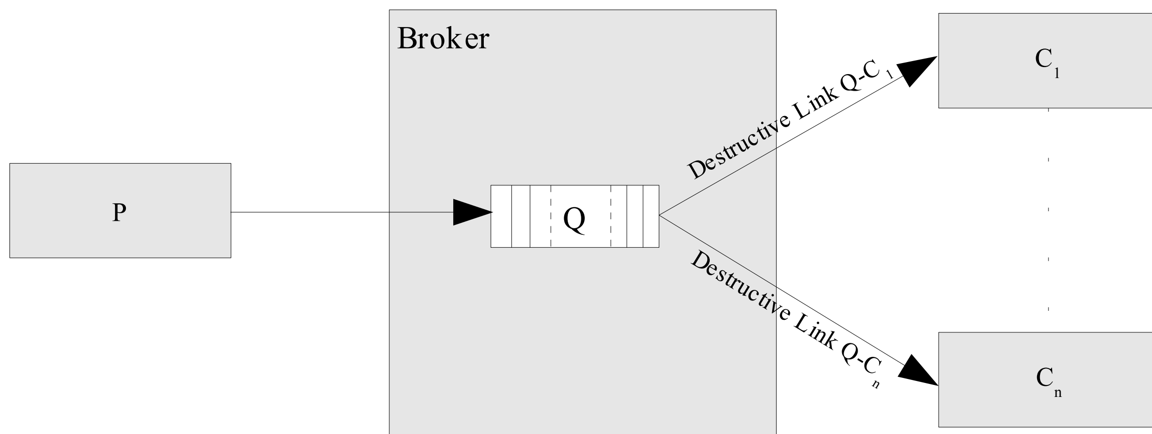
The Broker must be configured with a Queue "Q". The producing client opens a session with the broker. On the session it creates a link to send messages to address "Q". The producer then sends any messages over that link. On the consuming side, the consumer "C" also opens a sessions with the broker, and creates a destructive a link to receive messages from address "Q". As messages arrive in Q, they are sent to the consumer.

Since the destructive link from the Q to C has no filter, it will always point to the (single) head of the queue. Unless messages have been sent with differing priorities, this head will always be the oldest message in the queue.

Messages are removed from the queue one the consumer has informed the queue it has completed the processing of the message. If the session between the broker and he consumer is destroyed while there are messages that have been sent to the consumer but which have not yet been acknowledged as processed, then they will be returned to the queue.

7.1.2 Point-to-Point: Shared Work Queue

Extending the single producer/consumer model slightly we have the case where messages on the queue Q represent items of work, and the work can be done by any one of a number of consuming clients $C_1 ... C_n$. Each message should be processed by one and only one consumer.

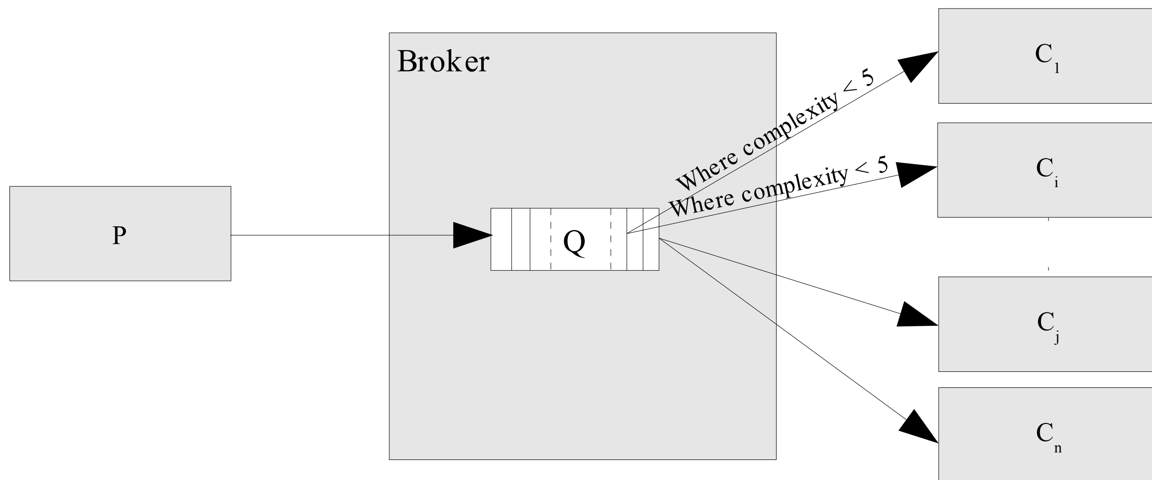


Again the Broker must be configured with a Queue "Q". The producing client opens a session with the broker. On the session it creates a link to send messages to address "Q". The producer then sends any messages over that link.

On the consuming side, each consumer "C_i" also opens a session with the broker, and creates a destructive a link to receive messages from address "Q". As messages arrive in Q, they are sent to the consumer. The fact that the link is "destructive" ensures that the message is "locked" on Q until the consumer it has been sent to informs the queue that the message has been processed. Thus each message is sent to one and only one queue. In general an AMQP does not mandate how a broker should balance the distribution of message between clients in such a scenario. It is expected that the broker should make efforts to evenly share out work between competing consumers, but should never wait to send a message to one consumer if there is an alternative consumer able to take the message immediately,

7.1.3 Point-to-Point: Shared Work Queue Using Filters

A slightly different take on the shared work queue is where we wish to place constraints on which consumer (or set of consumers) a particular message might go to. As an example let us assume that the messages contain some property "complexity" which is a numeric value; and that some consumers only wish to see messages where the complexity it less than 5, whereas other consumers will accept all messages.



Here the destructive links which have the filter "where complexity < 5" may be pointing at a head of the queue that in advance of the "true" head of the queue at which the other links are pointing.

7.1.4 Publish/Subscribe: Transient Pub/Sub

The principle differences between point-to-point and publish/subscribe style messages are that

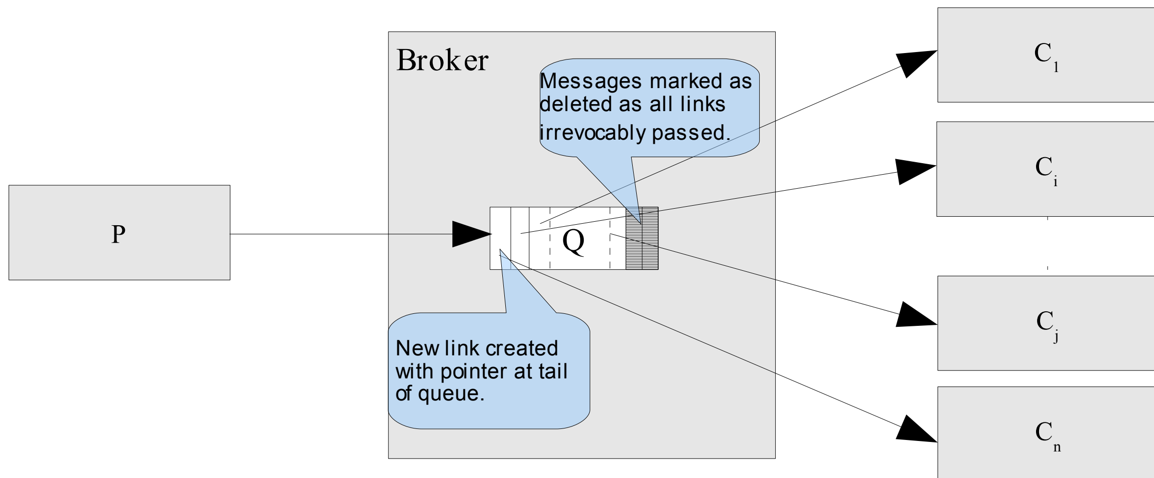
i) Messages should be sent to every subscriber, not individually distributed between them

ii) If there is no client listening for a message as it arrives in the broker, then it is conventionally discarded – not stored until such a client subscribes.

iii) Messages are not removed when one client has processed them, only when all active consumers have processed them.

By convention publish/subscribe systems are described in terms of "topics" rather than queues. AMQP has no distinct notion of "topics" - instead publish/subscribe is implemented using queues, although these queues have specific properties.

Within the AMQP broker model there are two ways that one may implement such a publish/subscribe topology. The simplest way is to simply define the topic as a queue to which no destructive links may be established. Each subscriber creates a non-destructive link to the queue. Depending on the link-configured desired behaviour may initial point to the head, the tail, or some intermediate point on the queue. Once all consuming links have irrevocably moved past a given message in the queue then that message is unreachable and can (depending on queue-configuration) be marked for deletion.



7.1.5 Publish/Subscribe: Durable Pub/Sub (TODO)

7.2 Multiple Broker Topologies

The topology used to connect Brokers and Clients on the Internet may be very different to the point-to-point and publish-and-subscribe messaging that the Client applications configure within and between Brokers. This is similar to Internet mail, where end users use email addresses perhaps unaware that the email service itself is built on top of IP addresses and port numbers and a handful of constituent protocols.

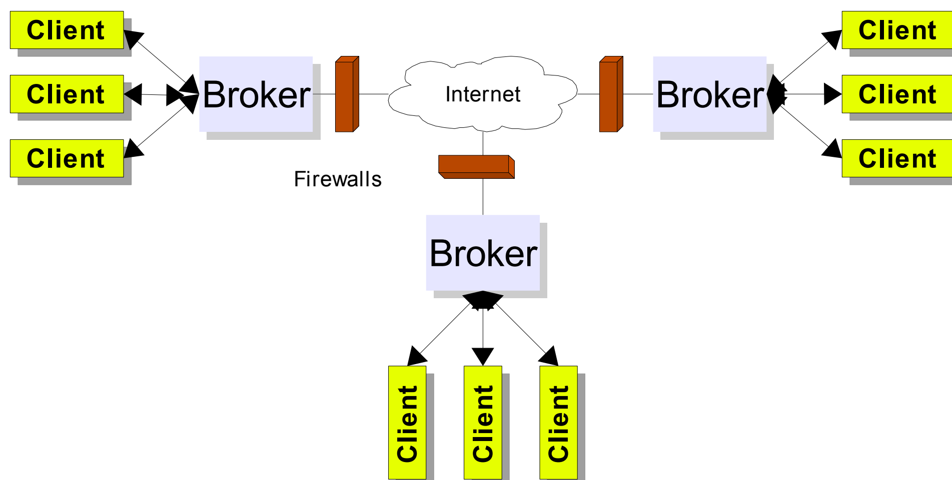


Illustration 1: Example AMQP Internet Topology

8 Glossary

8.1 Message (TODO)

8.2 Container

A Container represents a single physical or logical process which contains some number of AMQP nodes. Each Container has a globally unique identifier. For a given transport protocol this identifier may be associated with one or more network addresses. Authentication within AMQP is always with respect to a specific container. Examples of Containers are Brokers and Clients. Network connections are opened between Containers.

8.3 Node

A Node in the AMQP Network is a Message Source and/or a Message Sink. Messages are sent forwarded from one Node to another. Examples of Nodes are Queues, Message producers and Message Consumers. A Node are addressed by well-known names. Node names are scoped to the container in which the Node resides. A Node may have many names within the container.

8.4 Address

An AMQP address logically consists of the pair (Node Name, Container Id). Within a specific Container the Node name is in itself sufficient. More generally a network name or address may take the place of the container id. Thus the pair (name, example.com) would form an address. Such an address can be more conveniently written as name@example.com.

8.5 Link

A Link is a unidirectional transport for messages which is established between two Nodes. All Message transfers occur along Links. A Link may be between two Nodes in the same container, or Nodes in different containers. Link lifetime may be tied to the Nodes between which it transfers Messages, or to the lifetime of the Session on which it was created. Links may be either destructive (upon successful and acknowledged transfer the message is discarded by the source node) or non-destructive (a copy of the message is sent, but other links from the same Node will also be offered the same message). Nodes may have multiple incoming and/or multiple outgoing Links.

8.6 Session (TODO)

8.7 Queue (TODO)

8.8 Broker

A Broker is a Container in the AMQP Network that offers queuing services to other Containers (clients). Brokers are typically long-lived and would usually contain at least one Queue. Brokers that contain more than one Queue may be administratively configured to internally distribute messages to other Queues via the creation of broker-internal Links. Ordinarily a Broker requires authentication from any client nodes. In a transactional Session the broker functions as the transactional resource.

8.9 Client

Clients are end-user applications which connect to a Broker for the purpose of sending and/or receiving Messages. Clients may be on the same LAN or WAN within an organisation, or located in different organisations separated by firewalls and bureaucratic procedures. AMQP provides for a range of messaging semantics within an organisation, and connectivity to foreign organisations.

Presently AMQP does not prescribe the form of the Client API, only the capabilities of the Broker and how to communicate with it, but recommendations are given on how to map other technologies to AMQP.

8.10 Transaction (TODO)

End of Document