# Table of Contents

1  Overview........................................................................................................................................... 3
   1.1 What is a broker?......................................................................................................................... 3
   1.2 Relation to the other books........................................................................................................ 3
2  Client to broker protocol.......................................................................................................................... 5
   2.1 Addresses...................................................................................................................................... 5
   2.2 Messages...................................................................................................................................... 5
      2.2.1 Message format...................................................................................................................... 5
      2.2.2 Ordering................................................................................................................................ 5
      2.2.3 Priority................................................................................................................................... 6
      2.2.4 Message durability.................................................................................................................. 6
      2.2.5 TTL......................................................................................................................................... 6
   2.3 Links.............................................................................................................................................. 6
      2.3.1 Establishing and closing links................................................................................................. 6
      2.3.2 Re-establishing links............................................................................................................. 8
      2.3.3 Link lifetime........................................................................................................................... 8
      2.3.4 Errors....................................................................................................................................... 8
      2.3.5 Outcomes and settlement....................................................................................................... 9
      2.3.6 Delivery mode......................................................................................................................... 9
      2.3.7 Transfer failures..................................................................................................................... 9
      2.3.8 Publishing............................................................................................................................... 10
         2.3.8.1 Delivery mode.................................................................................................................. 12
      2.3.9 Flow control............................................................................................................................ 12
      2.3.10 Filtering................................................................................................................................. 12
      2.3.11 Subscribing and consuming................................................................................................. 12
         2.3.11.1 Subscribers..................................................................................................................... 13
         2.3.11.2 Delivery mode................................................................................................................ 14
         2.3.11.3 Consumers..................................................................................................................... 14
      2.3.12 Browsing............................................................................................................................... 15
      2.3.13 Dynamic sources and targets.............................................................................................. 15
3  Broker transactions............................................................................................................................... 16
   3.1 Transactional publish...................................................................................................................... 16
   3.2 Transactional accept....................................................................................................................... 18
   3.3 Transactional acquire..................................................................................................................... 20
1 Overview

1.1 What is a broker?
A broker is, generally speaking, a trusted intermediary. For a broker supporting AMQP 1.0 (henceforth just “broker”) this entails

- taking responsibility for messages on behalf of clients
- acting as a transaction resource, and possibly transaction co-ordinator, and
- routing and distributing messages

An AMQP broker is also a computer system, and as such, has operational aspects, some of which may be usefully standardised. For example, means of configuration and monitoring are set out in the Management specification.

Implementers need the behaviour of a server to be specified such that AMQP 1.0 can be retrofitted to legacy messaging solutions, or incorporated in non-traditional solutions. Applications need behaviour to be tightly-enough specified that they can reliably write useful applications that can be counted on to function correctly. These needs motivate our definition of a **client to broker protocol**, outlining requirements that can characterise the behaviour of existing systems, and allow general purpose AMQP 1.0 clients to interact with them; and, requirements for **broker transactions** in very much the same vein.

1.2 Relation to the other books
Books II and III define a type system and codec; an abstract protocol, using the type system, for transferring messages between peers and agreeing on the outcome (the “transfer protocol”); and a concrete protocol, using the codec, over TCP. Book IV introduces types for declaring behaviour at link sources and link targets, and specific outcomes useful for general-purpose messaging between peers. Book V specifies types for addressing transactional resources, and for declaring and discharging transactions.

This book adds to the messaging model the notion of a broker, and a model for trusted intermediation of message transfer. It supplements the messaging model with requirements for application and operational concerns; for example, persistence of messages, and ordering semantics.

In terms of the books mentioned above, a broker provides an TCP/IP server to service Connections, and keeps track of Sessions and Links created by the client. It resolves sources (for outgoing links) and targets (for incoming links), and implements a protocol specialised to client-broker interactions. It may also act as a transactional resource for certain operations.

A broker may also provide services for applications that are orthogonal to the messaging model; for example, authentication and federation. These are not discussed here.
Book VII - Messaging brokers
2 Client to broker protocol

The client to broker protocol covers the common scenarios which clients and brokers are communicating using AMQP 1.0. Central to these scenarios is the idea of the broker taking responsibility for messages on behalf of a downstream consumer, and its implications of persisting and buffering messages.

Where there are requirements of brokers, these generally follow the maxim that a broker must not lie. In other words, where the protocol defines types for declaring behaviour, the broker must use them to accurately indicate its behaviour. A broker may however tolerate, to some extent, clients misrepresenting or missing out information.

Many requirements also follow a “fail fast” principle; that is, any difference in the expectation of the client and what the broker is able to provide must be signalled by raising an error as soon as possible, as it indicates a mistake in the application logic or deployment.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

2.1 Addresses

A broker MUST resolve addresses given in link targets and link sources consistently with addresses used elsewhere – e.g., in a management interface. This is to ensure, for example, that an application can use an address generated by the broker as a reply-to header and expect any replies to appear along a link sourced at the address.

The simplest, though by no means only, interpretation of this is that source and target names are in a single namespace and each broker node has a single name by which it can be addressed.

However, we may also wish to hide the internal structure at the broker, and give nodes aliases, which can then be shared. In this case, it is the alias that is used as the address.

In circumstances in which the broker is obliged to generate a name, the name SHOULD be randomly generated; i.e., it should be hard to guess.

2.2 Messages

2.2.1 Message format

Book IV describes a message format that includes transport headers (important for message delivery) and message properties (immutable properties of the message). In general it is not necessary to discuss these here, other than to require that brokers MUST NOT change message properties; specific properties with implications for brokers are mentioned in the sections below.

A broker MAY be tolerant of malformed messages; e.g., may choose to accept messages missing sections, so far as it does not hinder the broker's operation. However, the broker MUST supply well-formed messages for clients downstream.

A broker MUST raise a session error for messages that misrepresent sections; e.g., by having incorrect format codes.

2.2.2 Ordering

Taking responsibility for messages on behalf of peers downstream implies FIFO buffers. However, there is no inherent requirement of ordering, except to say: Putting aside re-transfers, a broker SHOULD propagate messages over links in the order in which they are transferred from a producer.
2.2.3 Priority
The message header priority indicates that a message may “overtake” other messages with lower priority; i.e., be delivered before despite being published after.

Where priority is supported, messages MUST be ordered, for whichever guarantee of ordering is given, within equivalent priorities. Equivalence for priorities is described in book IV.

For example: Message A is sent before B is sent before C. If A, B, C are of equivalent priority, then they must be ordered A, B, C. However, if B and C are equivalent and higher priority than A, then the messages must be ordered either A, B, C or B, C, A or B, A, C.

2.2.4 Message durability
A broker should ensure no message is lost, unless it knows it does not need to. Sending a non-durable message effectively means “Please trade recoverability for speed”; it is a hint that the broker can optimise for this case, e.g., by settling an outcome straight away, avoiding using durable storage, and not requiring explicit settlement from consumers.

Brokers SHOULD “safely” store durable messages before settling transfers, in particular if the intermediary cannot deliver the message onwards immediately. A working definition of “safe” is “durable messages will survive software restarts”.

2.2.5 TTL
Brokers SHOULD respect ttl supplied in message headers. This means that a message must not be transferred along outgoing links after the TTL has elapsed; the message MAY be discarded at this point.

If a transmit-time is not also supplied, the broker is free to reckon TTL expiry from the time at which it received the message. If a transmit-time is supplied, a clock local to the broker MAY be used as an approximation for reckoning the TTL expiry.

2.3 Links
2.3.1 Establishing and closing links
The attach and detach frames are assertions of the link state as well as instructions. The value in the local field of these frames indicates the state known at the sender; the remote field indicates the sender’s knowledge of the state at the receiver. This implies some patterns in how the protocol is used to establish and close links:

<table>
<thead>
<tr>
<th>Field values</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>local=(A, B), remote=NULL</td>
<td>“There is a link here”. Used to establish a link.</td>
</tr>
<tr>
<td>local=NULL, remote=(X, Y)</td>
<td>“There is no link here”. Used to close a link, destroying it.</td>
</tr>
<tr>
<td>local=NULL, remote=NULL</td>
<td>“There is no link here, and I know of no link there”. This is probably</td>
</tr>
<tr>
<td></td>
<td>a reply to local=NULL, remote=(X, Y), or it may be used by a client that</td>
</tr>
<tr>
<td></td>
<td>has forgotten the remote state.</td>
</tr>
<tr>
<td>local=(A, B), remote=(X, Y)</td>
<td>“There is a link here and I know of a link there”. Usually sent as a</td>
</tr>
<tr>
<td></td>
<td>response to local=(X, Y), remote=NULL, or to re-establish a link.</td>
</tr>
</tbody>
</table>

Table 1: Meaning of null in local and remote fields

If a broker cannot honour a link as specified by a client, it MUST respond with either a null target in local (for links in which it is the receiver) or a null source in local (for links in...
which it is the sender). This is referred to as “refusing a link”, and should be regarded by the client as a failure to establish the link.

When establishing a new link, some fields are expected to be supplied definitively by the client; some are expected to be supplied by the client as an indication of its requirements of the broker; and some must be left null for the broker to supply definitively.

<table>
<thead>
<tr>
<th>Field</th>
<th>Supplied by</th>
</tr>
</thead>
<tbody>
<tr>
<td>options</td>
<td>May be supplied by the client. For some values, the broker may refuse the link; the broker may not add values.</td>
</tr>
<tr>
<td>name</td>
<td>Must be supplied by the client.</td>
</tr>
<tr>
<td>handle</td>
<td>Must be supplied by both peers (and will in general be different).</td>
</tr>
<tr>
<td>flow-state</td>
<td></td>
</tr>
<tr>
<td>unsettled-lwm</td>
<td>Must be supplied by client to indicate its current known state; may be null for a new, receiving link.</td>
</tr>
<tr>
<td>session-credit</td>
<td>Must be given by client to indicate its current known state.</td>
</tr>
<tr>
<td>transfer-count</td>
<td>Must be given by a sending client; may be omitted by a receiving client if the link is new. The broker must either echo or supply a value.</td>
</tr>
<tr>
<td>link-credit</td>
<td>Must be omitted by a sending client; may be supplied by a receiving client.</td>
</tr>
<tr>
<td>available</td>
<td>May be omitted.</td>
</tr>
<tr>
<td>drain</td>
<td>May be supplied by a receiving client. If omitted, the value is assumed to be “false”.</td>
</tr>
<tr>
<td>role</td>
<td>Must be supplied and consistent with local and remote.</td>
</tr>
<tr>
<td>local</td>
<td></td>
</tr>
<tr>
<td>source</td>
<td>Must be supplied by a sending peer for a new link; may be omitted otherwise. For a sending client, default-outcome and outcomes must be supplied; other fields may be left null.</td>
</tr>
<tr>
<td>target</td>
<td>Must be supplied by a receiving peer. A receiving client may leave any field null.</td>
</tr>
<tr>
<td>remote</td>
<td>Must be supplied by the broker and echo the local given in the request.</td>
</tr>
<tr>
<td>durable</td>
<td>May be supplied by the client. If absent, must be supplied by the broker.</td>
</tr>
</tbody>
</table>
Table 2: Link establishment fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expiry-policy</td>
<td>May be supplied by the client. If absent, must be supplied by the broker and be the value “session”.</td>
</tr>
<tr>
<td>timeout</td>
<td>May be supplied by the client. If absent, must be supplied by the broker and be the value “0”.</td>
</tr>
<tr>
<td>unsettled</td>
<td>May be supplied by the client if the link is not new.</td>
</tr>
<tr>
<td>transfer-unit</td>
<td>May be supplied by a sending client and in this case may be overruled by the broker¹. May be supplied by a receiving client; if absent, must be supplied by the broker.</td>
</tr>
<tr>
<td>max-message-size</td>
<td>As for transfer-unit.</td>
</tr>
<tr>
<td>error-mode</td>
<td>May be supplied by the client; if absent, must be supplied by the broker.</td>
</tr>
<tr>
<td>properties</td>
<td>May be supplied by either peer and may be different.</td>
</tr>
</tbody>
</table>

2.3.2 Re-establishing links
A client may wish to change the parameters of a link without closing it; e.g., to change the filter-set of the link. To do so, the client sends detach with the old parameters, then attach with the new parameters (in the example, a source with the new filter set, in the local field). The broker may of course refuse the link.

2.3.3 Link lifetime
In some circumstances a broker will need to close links itself; for example, if the source or target is destroyed. A client, therefore, must be prepared to receive detach frames from the broker. This is an exception to the general pattern of the client instructing the broker.

2.3.4 Errors
Errors MUST result in a link detach, session end or connection close.

Specifically,

- A broker MUST respond to malformed frames by sending close with an appropriate error and closing the underlying socket.
- A broker MUST respond to errors in begin by closing the session; i.e., sending end with an appropriate error, waiting for the corresponding end from the client.

How a broker responds to errors in attach, transfer, flow-control and disposition depend on the error-mode of the associated link. If the error-mode is end, the broker MUST close the session, i.e., send end and wait for the corresponding end; and if detach, the broker MUST detach the link.

¹transfer-count is calculated using the value of transfer-unit. In some circumstances, clients may wish to send attach then transfer without waiting for a corresponding attach, and thereby will not have the authoritative value for transfer-unit with which to calculate the transfer-count sent with the transfer. In this case, the client SHOULD send null for the transfer-unit; a broker MAY then choose to honour the transfer, supply the transfer-unit with its response, and expect the client to recalculate the transfer-count subsequently.
Invoking the process described immediately above is referred to in the following as “raising an error”.

2.3.5 Outcomes and settlement
A broker in general acts to take responsibility for messages by settling outcomes; either from senders, by accepting transfers with a settled state, or from receivers, by settling the posited outcome.

Usually an outcome will be settled by advancing the low water mark (unsettled-lwm, sent in transfer and flow). In some circumstances an explicit disposition exchange is required, giving the outcome and asserting that the transfer is settled. A disposition MAY be sent if a state is not given in the transfer; otherwise, the outcome is that given as the default-outcome.

A peer MUST treat conflicting statements of the outcome for a transfer as an error. For example, if a client sends an accepted outcome then a rejected outcome for a particular transfer, the broker must close the link or session, reporting an error; and vice versa.

The exception to this is the provisional outcome given for a transfer in the scope of a transaction, which may change once the transaction is discharged.

2.3.6 Delivery mode
In certain circumstances a client may need to indicate to the broker its required delivery semantics, where these are not implied by either the link default-outcome and outcomes fields, or the transfer state and settled fields. The client may also wish to explicitly allow or deny the broker this assumption. To do this, the client uses one of the options “delivery-mode” or “opt-delivery-mode”, both symbols used as keys in the options map of attach.

Using “delivery-mode” as the key forces the broker to support the mode or refuse the link. Using “opt-delivery-mode” indicates that the client will cope if the mode is not followed. Both may be supplied for a link. In the following delivery-mode, unquoted, refers to an link established with the “delivery-mode” option specifically.

The possible values for both keys are the following symbols: “at-most-once”, “at-least-once”, “exactly-once”. The requirements indicated by each value will be discussed in 2.3.8.1 and 2.3.11.2.

2.3.7 Transfer failures
The messaging model admits varieties of responsibility transfer. Central to this is the idea of settlement; that is, agreement on a particular outcome for a transfer. However, the receiver of a link can disappear without responding and not come back (i.e., time out) without the outcome being settled; we need to account for what happens to messages that have been committed for transfer over that link, but which are not known to have been transferred successfully because the outcome has not been settled.

The source type in book IV defines an default-outcome field, which gives the assumed outcome for transfers left unsettled.

Generally speaking, there are four options for dealing with unsettled messages:

1. Try to deliver the message to some other link from the node, or re-queue it; the default-outcome is released
2. Send the message to a “Dead letter queue”; default-outcome of reject
3. Return the message to the publisher, given a suitable return route; default-outcome of reject
4. Abandon the delivery; in this case, the default-outcome is accepted.
A client can supply null in *default-outcome*, indicating that it agrees with whichever *default-outcome* the broker supplies; or, it can supply a *default-outcome* to indicate its expected behaviour.

A broker **MUST** enforce its policy or configuration where present by refusing links that do not agree on the *default-outcome*. Such a policy or configuration can admit more than one possible default-outcome; for example, the broker might allow a default-outcome of *release* for consumers that require "at least once" delivery, or *accept* for consumers requiring "at most once" delivery.

### 2.3.8 Publishing

Publishing a message requires a link established with the client as sender and broker as receiver. Broadly speaking, there are two styles of arranging this:

- one-off; i.e., establishing a link and transferring one (or perhaps a small handful) message, then closing the link; and,
- continuous, that is, establishing a long-lived link and transferring many messages.

It is desirable, for the first scenario especially, to be able to establish a link, transfer a message, and possibly even close the link without having to wait for a response from the broker. For this reason, a broker **MAY** choose to tolerate incorrect flow state for transfers along new links, as described in 2.3.1.

If a client wishes to receive acknowledgement for a transfer, it is enough to supply a *state* (or rely on the default outcome) and the value *false* for *settled*. The broker is then obliged to settle the transfer.

```python
Attach(options=None, name="publish-link-1234", handle=4, flow_state=FlowState( unsettled_lwm=1, session_credit=10, transfer_count=0, link_credit=None, available=None, drain=False), role=False, local=Linkage( source=Source( outcomes=['amqp:accept:map'], default_outcome=Accept()), # We are a sender. # Here we are saying that the broker # can simply advance the LWM to # indicate that it has accepted # messages, since there is only one # possible outcome. This is # effectively "at most once", since # a dropped connection will settle # transfers to the default outcome. target=Target(address="my_topic")), remote=None, transfer_unit=None, max_message_size=0, error_mode="detach") # Session-level low water mark. Here # we are starting from scratch, so we # say the LWM is the next transfer-id.

Attach(options=None, name="publish-link-1234", handle=5, flow_state=FlowState( unsettled_lwm=1, session_credit=10, transfer_count=0, # The broker has its own handle, which # we have to recognise. # We told the broker these, and # nothing has been sent yet, so it )```

---

<table>
<thead>
<tr>
<th>Source</th>
<th>Target Address</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>amqp:accept:map</td>
<td>my_topic</td>
<td>We are a sender. Here we are saying that the broker can simply advance the LWM to indicate that it has accepted messages, since there is only one possible outcome. This is effectively &quot;at most once&quot;, since a dropped connection will settle transfers to the default outcome.</td>
</tr>
</tbody>
</table>
# remains the same.

# The broker gives us some initial credit, with which to transfer messages.

# The broker agrees on the linkage.
# If it did not, it would have to refuse the link by giving a null target here.

# The broker tells us its defaults where we have not supplied values.

# We now have a link, targeting "my_topic". We use the handle 4 and the broker uses the handle 5.
# We'll transfer a message. This is our alias for the link.
# This is our alias for the link.

# The LWM refers to this transfer. By sending an unsettled transfer, we are reducing the session credit.
# We reduce our credit by the "size" of this transfer.

# delivery-tag is arbitrary, but transfer-id is a serial number. It corresponds to our LWM above.
# We will be waiting for the broker to settle this. We don't need to supply a state, since there is only one outcome possible and we're not using a transaction.
# Our message (elided here)

# The broker is obliged to respond as soon as possible, since we said batchable is false. It does not need to send a disposition frame, though; it can get away with a flow frame.

# The broker's handle for the link

# The broker advanced the LWM to indicate that it considers the transfer above to be settled. We now consider the transfer to have the default-outcome.

# It also put the session credit back up to 10, since a transfer has now been settled.
Table 3: Settling a transfer with the low water mark

If the transfer has batchable as true, the client is indicating that the broker can delay before settling, in order to settle many transfers at once. If batchable is false, the broker SHOULD settle the transfer as soon as possible, as this could indicate for example that the client is blocking on the transfer being settled.

In any case, the broker SHOULD NOT hold up a publisher by exhausting its session credit without settling transfers.

In some circumstances the client may also wish to require the broker to send a disposition frame; for instance, if there is a choice of outcome for the broker to make. In this case, the transfer will have a state with no outcome given, and the client should specify the outcomes during link establishment.

Often, however, a publisher will not require acknowledgement from the broker, in which case it will supply an outcome and assert that it is settled in the transfer frame. This may or may not advance the low water mark.

2.3.8.1 Delivery mode

The delivery-mode values for publishing are “exactly-once” and “at-least-once”.

If a delivery-mode of “exactly-once” is supported, the broker SHOULD de-duplicate transfers as identified by delivery-tag. The broker MAY ask the client for confirmation of transfer outcome (as explained in book III). The client SHOULD transfer with an unsettled outcome.

If a delivery-mode of “at-least-once” is supported, the broker SHOULD NOT indicate an unsettled outcome for a transfer to the client; i.e., it should not ask the client for confirmation of transfer outcome. The client SHOULD transfer with an unsettled outcome.

2.3.9 Flow control

Book III defines a credit-based flow control mechanism. For a broker there are two contracts implied: for incoming links, and for outgoing links.

The contract with regard to incoming links is that the broker MUST NOT issue more credit than it can honour. How the broker distributes credit among incoming links will differ depending on policy and configuration; however, for a broker to participate reliably in an AMQP network it MUST NOT, taken across all incoming links, over-represent its capacity for receiving messages.

However, a broker MAY choose to tolerate clients that do not strictly follow flow control. In particular, a broker can deliberately omit transfer-count in flow-state, to indicate to a producer that it is not currently enforcing flow control.

With regard to outgoing links, a broker must fulfil its obligations as instructed by the client. In other words, it MUST NOT transfer more messages over a link than it has credit on the link.

2.3.10 Filtering

Book IV introduces filters and filter sets. In general, brokers are not required to support filters. A broker MUST refuse an outgoing link if it cannot support the filter set given when establishing a link.

2.3.11 Subscribing and consuming

Book IV defines distribution-mode, which is used to indicate the desired (by a client) or determined (by a broker) behaviour of a source.

A broker is not in general required to support the distribution-mode supplied by a client; if the broker will not fulfil a distribution-mode supplied by a client, it MUST refuse the link.
Commonly, for a given address the broker will either have a policy of distributing to all outgoing links, in which case it accepts sources with copy; or, a policy of distributing each message to one outgoing link exclusively, in which case it accepts sources with move. Elsewhere these are called “topics” and “queues” respectively; we will adopt these terms for convenience.

To distinguish between the kinds of outgoing link, we will say that topics have subscribers, and queues have consumers. These are also used, where unambiguous, to refer to the client establishing such a link.

2.3.11.1 Subscribers

Subscribers use outcomes as transfer acknowledgement; as such, only accept has a defined meaning as a default-outcome, which is that the transfer in question must not be made along the link again. The semantics in the case of reject, release and modified outcomes is left undefined.

There are two modes to settle a transfer; in the first, the subscriber wishes to explicitly acknowledge each transfer with a disposition. In the second, the subscriber will acknowledge transfers by advancing the unsettled-lwm in flow frames.

Supplying a single outcome in the outcomes field of attach means the broker can assume that state for messages. In this scenario, the broker SHOULD transfer messages with the indicated state; the subscriber can either send a disposition frame or simply advance the unsettled-lwm in a flow frame in order to acknowledge the transfer.

It is worth observing that a broker can go further and transfer with a settled outcome if, for example, the outcomes for a link consists only of “amqp:accept:map” and its default-outcome is accept, and the link is bound to the lifetime of a session.

The following pseudo-transcript illustrates subscription. An open connection and session are assumed.

```
Attach(name="subscribe-link-1234",
    handle=15, # We are the receiver.
    role=True,
    local=Linkage(target=None, # The target is unimportant.
        source=Source(
            address="my_topic", # Names the source
            dynamic=None, # It's a well-known address.
            distribution_mode="copy", # We are expecting this to be a
            default_outcome=Accept(), # subscription
            outcomes=["amqp:accept:map"], # Messages implicitly settled should be
                                    # accepted, and
            remote=None)) # we will only ever accept transfers.

Attach(name="subscribe-link-1234",
    handle=11, # The broker's handle for the link.
    role=False, # The broker is the sender.
    local=Linkage(target=None, # The broker confirms the link target
        source=Source(
            address="my_topic", # and source, to show the link is
            remote=None)) # established
```
Table 4: Subscribing

2.3.11.2 Delivery mode
The delivery modes for subscribers are “at-most-once” and “at-least-once”.

If a delivery mode of “at-most-once” is supported, the broker SHOULD assume transfers are settled with the default-outcome when sending.

If a delivery-mode of “at-least-once” is supported, the broker MUST NOT assume that transfers are settled when sending. The client MAY respond with an unsettled outcome, indicating that it requires confirmation of the outcome from the broker.

2.3.11.3 Consumers
Consumers use outcomes as instructions. Specifically,

- **accepted** instructs the broker to not transfer the message again;
- **release** instructs the broker to redistribute the message;
- **reject** instructs the broker to invoke rejected message handling e.g., sending the message to a dead letter queue;
- **modified** instructs the broker to reconsider the message, with header alterations, for distribution.

Brokers MUST indicate the outcomes available in the `outcomes` field of the `source`, when responding to the client during link establishment.

A client MAY itself specify a set of the outcomes during link establishment; in this case, the client is specifying that it will only use certain outcomes. If this is not a subset of those supported by the broker, it MUST refuse the link. If it is, the broker MUST echo the outcomes in its response.

If the client then gives an outcome not in the set, the broker MUST raise an error.

The following pseudo-transcript shows a consumer establishing a link. As above, it assumes an open connection and session.
| name="consume-link-1234",
| handle=11,
| role=False,
| local=Linkage(target=None,
| source=Source(address="my_queue",
| dynamic=None,
| distribution_mode="move",
| default_outcome=Release(),
| outcomes=["amqp:accept:map"])),
| remote=Linkage(target=None,
| source=Source(address="my_queue",
| dynamic=None,
| distribution_mode="move",
| default_outcome=Release(),
| outcomes=["amqp:accept:map"])),
# (For the broker to supply)

Attach(name="consume-link-1234",
| handle=11,
| role=False,
| local=Linkage(target=None,
| source=Source(address="my_queue",
| dynamic=None,
| distribution_mode="move",
| default_outcome=Release(),
| outcomes=["amqp:accept:map"])),
| remote=Linkage(target=None,
| source=Source(address="my_queue",
| dynamic=None,
| distribution_mode="move",
| default_outcome=Release(),
| outcomes=["amqp:accept:map"])),
# This is an echo of the local
# field supplied by the client.

Table 5: Consuming

2.3.12 Browsing
Various use cases require the ability to receive messages from a node without interacting with its distribution of messages. This is partially encoded in the protocol by a link source specifying a distribution-mode of copy (when it would otherwise be expected to be move); move meaning that the link is considered when distributing messages, and copy meaning in this case that it is considered in addition to distributing messages.

The semantics for outcomes are the same as for other links specifying copy in the source.

2.3.13 Dynamic sources and targets
A broker may support the creation of dynamic sources or targets, or both. If so, a client MAY use the dynamic field to request such a source or target. Aside from the lifetime given in dynamic, the nature of the source or target is undefined.

The lifetime defines the earliest point at which the dynamic source or target may be destroyed. Where the lifetime is bounded by an explicit action of the client (e.g., link closure using detach), the destruction SHOULD be enacted before the broker responds to the action (in this example, before it sends the corresponding link detach).
3 Broker transactions

A broker MAY act in the role of transactional resource manager, enacting units of work durably and atomically. What “durably” entails depends on the broker policy or configuration and the source or target; a working principle is that the relevant state has been “safely” stored; e.g., to disk. “Atomically” has the usual sense, that is, the entire unit of work is completed or none of it is.

A broker MAY also act as a transaction co-ordinator if it at least supports local transactions. In this case it MUST recognise the transaction coordinator target as defined in book V. When establishing a link to the coordinator target, a client MAY omit the source field.

Settlement and transactions are related; peers MUST NOT settle an outcome before the transaction with which it is associated has been discharged. In protocol terms, this means that no transfer state can have both be settled and have a txn_id, except for the transfer state of a transaction discharge.

A broker MAY indicate a provisional outcome in the context of a transaction by sending disposition with an unsettled transfer-state. If this is done, the broker MUST honour that indication with its settled outcome once the transaction is discharged, or fail to discharge the transaction.

Thus, if a receiver sends an unsettled transfer-state in the context of a transaction, it MUST NOT be treated as part of a settlement exchange; e.g., the sender MUST also wait until the transaction is discharged before settling transfers.

In the case of a failed transaction, it is understood that associated transfers and dispositions are rolled back. It is not necessary to exchange transfer-state. Flow-control state is not rolled back; e.g., transfers that are part of a failed transaction still consume credit.

3.1 Transactional publish

In this example we show a protocol exchange for a client transactionally publishing to a broker. The connection and session establishment are assumed, and flow control and other irrelevant details are elided.

Generally the frames are asynchronous; however, there are certain points at which the client has to wait for a response in order to proceed; e.g., when declaring a transaction, the client as transaction controller needs the transaction ID from the response in order to use it with the transfer that follows. Below, the frames are shown in request/response order to aid reading.

```plaintext
Attach(name="txn-link-1234",  # First we establish a link to
       handle=4,                  # the transaction coordinator.
       role=False,               # Our handle for the link.
       local=Linkage(            # We are the sender.
           source=None,
           target=Coordinator(    # We are asking for a link to
               capabilities=\["amqp:local-transactions"]),
           remote=None))          # the transaction coordinator,

Attach(name="txn-link-1234",  # The broker echoes the name,
       handle=5,               # and gives its handle.
       role=True,              # Broker is the receiver.
       local=Linkage(          # The broker says that it
           source=None,
           target=Coordinator(   # does have a transactional
               capabilities=\["amqp:local-transactions"],
           remote=None))        # coordinator, and that it
```
"amqp:distributed-transactions", "amqp:promotable-transactions"}],
remote=Linkage(
  source=None,
  target=Coordinator(
    capabilities="amqp:local-transactions")))
# modes.

# The broker echoes back our statement of the linkage.
# Now we have a link to the transaction co-ordinator, with handle 4 for send and handle 5 for receive.
# Our handle to the coordinator
# This is arbitrary
# This is a serial number alias
# We expect confirmation from the broker.
# Our handle for this link
# Our handle to the coordinator
# This is arbitrary
# This is a serial number alias
# We expect confirmation from the broker.

Transfer(handle=4,
  delivery_tag="begin321",
  transfer_id=16,
  settled=False,
  state=TransferState(outcome=Accepted()),
  fragments=[Fragment(
    format_code=4,
    first=True,
    last=True,
    payload_offset=0,
    payload=Declare(global_txn_id=None))])
# Let the broker create a local transaction ID
# We can now use that ID to associate frames with the transaction.
# The declaration is successful, and the created transaction ID is "txn1234"

Disposition(role=True,
  extents=[Extent(
    first=16,
    last=16,
    handle=5,
    settled=True,
    state=TransferState(outcome=Accepted(),
      txn_id="txn1234")))
# The broker is the receiver
# Alias of transfer just made
# The broker's handle
# The transfer state is decided

Attach(name="my-link-1234",
  handle=11,
  role=False,
  local=Linkage(
    source=Source(outcomes="amqp:accepted:map"),
    default-outcome=Accept()),
  target=Target(
    address="my_topic"),
  remote=None))

Attach(name="my-link-1234",
  handle=12,
  role=True,
  local=Linkage(
    source=Source(outcomes="amqp:accepted:map"),
    default-outcome=Accept()),
  target=Target(address="my_topic"),
  remote=Linkage(
    source=Source(outcomes="amqp:accepted:map"),
    default-outcome=Accept()),
  target=Target(address="my_topic")))

Transfer(handle=11,
  # Our handle for this link
In this example, we demonstrate accepting a transfer within a transaction.

Table 6: Transactional publish

### 3.2 Transactional accept

In this example, we demonstrate accepting a transfer within a transaction.
The broker says that it does have a transactional coordinator, and that it supports these transactional modes.

The broker echoes back our statement of the linkage.

Now we have a link to the transaction co-ordinator, with handle 4 for send and handle 5 for receive.

Establish a link to consume from "my_queue".

Immediately issue credit, so the broker can transfer messages straight away.

Establish a link to consume from "my_queue".

Immediately issue credit, so the broker can transfer messages straight away.

At this point, we have a message transferred to us.

Now we'll accept in, using a transaction.
settled=False,
state=TransferState(outcome=Accepted()),
fragments=[Fragment(
    format_code=4,
    first=True,
    last=True,
    payload_offset=0,
    payload=Declare(
        global_txn_id=None))]

# We expect confirmation
# from the broker.
# amqp-data
# NB: in general, payloads are
# encoded and sent as a binary
# Let the broker create a local
# transaction ID
Disposition(role=True,
    extents=[Extent(
        first=16,
        last=16,
        handle=5,
        settled=True,
        state=TransferState(
            outcome=Accepted(),
            txn_id="txn2345"))])

# The broker is the receiver
# Alias of transfer just made
# The broker's handle
# The transfer state is decided
# The declaration is
# successful, and the created
# transaction ID is “txn2345”
Disposition(role=True,
    extents=[Extent(first=0,
        last=0,
        handle=11,
        settled=False,
        state=TransferState(
            outcome=Accepted(),
            txn_id="txn2345"))])

# We have to send an explicit
# disposition, since we need to
# supply the txn_id in the
# state.
Transfer(handle=4,
    delivery_tag="commit654",
    transfer_id=18,
    settled=False,
    state=TransferState(txn_id="txn2345"),
    fragments=[Fragment(
        format_code=4,
        first=True,
        last=True,
        payload_offset=0,
        payload=Discharge(fail=False))])

# Our transaction was committed.
# Note the broker cannot send
# both dispositions in one,
# since they are referring to
# transfers in different
# directions.
Disposition(role=True,
    extents=[Extent(first=18,
        last=18,
        handle=5,
        settled=True,
        state=TransferState(
            outcome=Accepted()))])

Disposition(role=False,
    extents=[Extent(first=0,
        last=0,
        handle=17,
        settled=True,
        state=TransferState(
            outcome=Accepted()))])

# The broker settles the state
# of the transfer. It must wait
# until after sending the
# transaction settlement.

Table 7: Transactional accept

3.3 Transactional acquire

This example shows the transactional acquisition of a message. This is distinct from
transactional accept: acquiring a message means that it is unavailable for other
consumers, and committing does not confer an outcome; whereas committing a
transactional accept does indeed accept the transfer. Rolling back a transactional
acquisition means that the message is available again; whereas rolling back a 
transactional accept means that the transfer simply goes back to its previous state, but 
is still acquired.

```python
Attach(name="txn-link-1234",
    handle=4,
    role=False,
    local=Linkage(
        source=None,
        target=Coordinator(
            capabilities=["amqp:local-transactions"],
            remote=None)
    )
    # First we establish a link to
    # the transaction coordinator.
    # Our handle for the link.
    # We are the sender.
    # We are asking for a link to
    # the transaction coordinator,
    # and for it to support local
    # transactions.
    # The broker echoes the name,
    # and gives its handle.
    # Broker is the receiver.
    # The broker says that it
    # does have a transactional
    # coordinator, and that it
    # supports these transactional
    # modes.
    # The broker echoes back our
    # statement of the linkage.
    # Now we have a link to the
    # transaction co-ordinator, with
    # handle 4 for send and handle 5
    # for receive.
    # Establish a link to consume
    # from "my_queue".
    # Don't issue credit, because we
    # will want to associate a
    # txn_id with the credit.

Attach(name="txn-link-1234",
    handle=5,
    role=True,
    local=Linkage(
        source=None,
        target=Coordinator(
            capabilities=["amqp:local-transactions",
            "amqp:distributed-transactions",
            "amqp:promotable-transactions"],
            remote=None)
    )
    # The broker echoes back our
    # statement of the linkage.
    # Now we have a link to the
    # transaction co-ordinator, with
    # handle 4 for send and handle 5
    # for receive.
    # Establish a link to consume
    # from "my_queue".
    # Don't issue credit, because we
    # will want to associate a
    # txn_id with the credit.

Attach(name="my-link-1234",
    handle=11,
    role=True,
    transfer_unit=0,
    flow_state=FlowState(
        transfer_count=None,
        link_credit=0),
    local=Linkage(
        source=Source(
            address="my_queue",
            distribution_mode="move",
            outcomes=["amqp:accepted:map"],
            default_outcome=Release()),
        target=Target()),
    remote=None)
    # Establish a link to consume
    # from "my_queue".
    # Don't issue credit, because we
    # will want to associate a
    # txn_id with the credit.

Attach(name="my-link-1234",
    handle=17,
    role=False,
    local=Linkage(
        source=Source(
            address="my_queue",
            distribution_mode="move",
            outcomes=["amqp:accepted:map"],
            default_outcome=Release()),
        target=Target(address="my_topic")),
    remote=Linkage(
        source=Source(
            address="my_queue",
            distribution_mode="move",
            outcomes=["amqp:accepted:map"],
            default_outcome=Release()),
        target=Target(address="my_topic"),
        remote=None)
    # Establish a link to consume
    # from "my_queue".
    # Don't issue credit, because we
    # will want to associate a
    # txn_id with the credit.
```
target=Target()

Transfer(handle=4,
   delivery_tag="begin765",
   transfer_id=32,
   settled=False,
   state=TransferState(outcome=Accepted()),
   fragments=[Fragment(
         format_code=4,
         first=True,
         last=True,
         payload_offset=0,
         payload=Declare(
                global_txn_id=None))])

Disposition(role=True,
   extents=[Extent(
         first=32,
         last=32,
         handle=5,
         settled=True,
         state=TransferState(outcome=Accepted(),
              txn_id="txn5432"))])

Flow(handle=5,
   options="(txn-id": "txn5432"),
   flow_state=FlowState(
      link_credit=1,
      drain=True))

Transfer(handle=17,
   delivery_tag="my-link-1234-0",
   transfer_id=0,
   settled=False,
   state=TransferState(txn_id="txn5432"),
   fragments=[...])

Transfer(handle=4,
   delivery_tag="commit765",
   transfer_id=33,
   settled=False,
   state=TransferState(txn_id="txn5432"),
   fragments=[Fragment(
# Transaction accepted. Now we have acquired the message.

# If we had rolled back, the message would have been made available again (possibly resulting in it being sent on our incoming link again, should we supply more credit).

Table 8: Transactional acquisition