# The Aqueduct Protocol: Reference Specification

Spec version: 0.0.0-AFTER (**Warning: Non-unique designation for draft version.**)

Note: It is recommended to read `OVERVIEW.md` first.

## 1 § Encoding low-level primitives

### 1.1 § Endianness

All values are encoded little-endian.

### 1.2 § Varint

A variable length unsigned integer is encoded as a sequence of 1 or more bytes, where the lowest 7 bits of each byte encode the next lowest 7 bits of the integer, and the highest bit of each byte is 1 if there are additional bytes in the sequence.

TODO: there is a discrepancy between this and my code

### 1.3 § Varbytes

A variable length sequence of bytes is encoded as a varint conveying the number of bytes, followed by the bytes.

### 1.4 § Varranges

A variable length sequence of unsigned integer ranges is encoded as a varbytes containing a sequence of varints. The varints alternate between the length of the next range and the length of the gap before the next range. All varints except the first and last must be non-zero.

### 1.5 § Header data

Header data is encoded as a varbytes containing a sequence of multiple inner varbytes. The number of inner varbytes is always a multiple of 2. Each pair of 2 inner varbytes represents a key/value pair.

### 1.6 § Chanid

A channel ID uniquely identified a channel within a connection. It is a bit field encoded as a varint. Its bits, from lowest to highest, are:

1. CREATOR (1 bit): 0 if the client created the channel, 1 if the server created the channel.
2. SENDER (1 bit): 0 if the client owns the sender half, 1 if the server owns the sender half.
3. ONESHOT (1 bit): 0 if the channel is multishot, 1 if the channel is oneshot.
4. INDEX (61 bits): An unsigned integer, assigned sequentially within the space defined by the other 3 bits, via an integer counter owned by CREATOR.

Naturally, the entrypoint chanid is all zeroes. Thus, the index counter for that index space starts at 1. All other index counters start at 0.

## 2 § Frames

### 2.1 § Receiving frames

Aqueduct utilizes QUIC unidirectional streams and QUIC unreliable datagrams, both of which contain a sequence of frames. An endpoint must read and process these frames. A frame

sequence always follows the pattern:

1. Zero or more of the following:
    1. Any of the following frame types: `VERSION`, `ACK_VERSION`, `CONNECTION_HEADERS`.
2. Zero or one of the following:
    1. One `ROUTE_TO` frame.
    2. Zero or more frames of types other than the following: `VERSION`, `ACK_VERSION`, `CONNECTION_HEADERS`, `ROUTE_TO`.

The `ROUTE_TO` frame contains a chanid that subsequent frames in the sequence pertain to. The endpoint must read process the subsequent frames sequentially in the context of its state for that channel.

If a stream is reset, the endpoint ignores any partially received frame.

### 2.2 § Frame types

The following frame types exist, and they are encoded as such:

#### 2.2.1 § `VERSION`

1. MAGIC BYTES: The bytes [239, 80, 95, 166, 96, 15, 64, 142].
2. HUMAN TEXT: The ASCII string "AQUEDUCT".
3. VERSION: A varbytes containing the ASCII string "0.0.0-AFTER".

#### 2.2.2 § `ACK_VERSION`

1. TAG: The byte 1.

#### 2.2.3 § `CONNECTION_HEADERS`

1. TAG: The byte 2.
2. CONNECTION_HEADERS: Header data.

#### 2.2.4 § `ROUTE_TO`

1. TAG: The byte 3.
2. CHANNEL: A chanid.

**2.2.5 § `MESSAGE`** Contextual: ROUTE_TO.CHANNEL.SENDER must be the frame sender.

1. TAG: The byte 4.
2. MESSAGE_NUM: A varint.
3. MESSAGE_HEADERS: Header data.
4. ATTACHMENTS: A varbytes containing a sequence of the following:
    1. CHANNEL: A chanid with CREATOR equal to frame sender.
    2. CHANNEL_HEADERS: Header data.
5. PAYLOAD: A varbytes.

**2.2.6 § `SENT_UNRELIABLE`** Contextual: ROUTE_TO.CHANNEL.SENDER must be the frame sender.

1. TAG: The byte 5.
2. COUNT: A varint.

**2.2.7 § `FINISH_SENDER`** Contextual: ROUTE_TO.CHANNEL.SENDER must be the frame sender.

1. TAG: The byte 6.
2. SENT_RELIABLE: A varint.

**2.2.8 § `CANCEL_SENDER`** Contextual: ROUTE_TO.CHANNEL.SENDER must be the frame sender.

1. TAG: The byte 7.

**2.2.9 § `ACK_RELIABLE`** Contextual: ROUTE_TO.CHANNEL.SENDER must be the frame receiver.

1. TAG: The byte 8.
2. RANGES: A varbytes containing a sequence of varints. The number of varints is even and non-zero. Every varint except the first must be non-zero.

**2.2.10 § `ACK_NACK_UNRELIABLE`** Contextual: ROUTE_TO.CHANNEL.SENDER must be the frame receiver.

1. TAG: The byte 9.
2. RANGES: A varbytes containing a sequence of varints. There must be more than zero varints. All varints must be nonzero, except the first if there are more than one.

**2.2.11 § `CLOSE_RECEIVER`** Contextual: ROUTE_TO.CHANNEL.SENDER must be the frame receiver.

1. TAG: The byte 10.

**2.2.12 § `FORGET_CHANNEL`** Contextual: ROUTE_TO.CHANNEL.CREATOR must be the frame sender.

1. TAG: The byte 11.

**2.2.13 § `DEQUEUED`** Contextual: ROUTE_TO.CHANNEL.SENDER must be the frame receiver.

1. TAG: The byte 12.
2. NUM_BYTES: A varint.

## 3 § Protocol operation

Both sides of the connection maintain a set of sender state machines and receiver state machines, each associated with a channel ID for which the endpoint is the sender / receiver. When the connection initializes, there is only a single sender state machine on the client and receiver state machine on the server, for the entrypoint channel.

### 3.1 § Version

Both sides must begin all frame sequences with a `VERSION` frame until they receive an `ACK_VERSION` frame. When an endpoint first receives a `VERSION` frame it must send an `ACK_VERSION` frame in a stream. If an endpoint receives a frame sequence that does not begin with a `VERSION` frame before it has sent an `ACK_VERSION` frame it must close the connection without processing subsequent frames.

### 3.2 § Connection headers

When the connection starts, the client must send the server a `CONNECTION_HEADERS` frame in a stream, containing the client's connection headers. When the server receives this, it must send back a `CONNECTION_HEADERS` frame in a stream, containing the server's connection headers. Each side must wait to process any frames other than `VERSION` frames until it has received a `CONNECTION_HEADERS` frame. Each side must only send a `CONNECTION_HEADERS` frame once.

### 3.3 § Sender-side operation

When an application sends a message via a sender handle, the endpoint sends a `MESSAGE` frame. In ORDERED mode, the endpoint maintains a single stream to send all these frames for the channel. In UNORDERED mode, the endpoint creates a new stream to send each of these frames. In UNRELIABLE mode, the endpoint sends each of these frames in an unreliable datagram, unless the message is too large to fit in an unreliable datagram, in which case it sends it in a stream.

Each channel has two different spaces of MESSAGE_NUM values: one for messages sent reliably in streams, and one for messages sent unreliably in datagrams. These numbers are assigned sequentially starting at zero within their spaces.

Shortly after an endpoint sends messages unreliably, it must send an a `SENT_UNRELIABLE` frame on a stream counting the number of additional messages sent unreliably on that channel.

When an application finishes a channel via a sender handle, the endpoint sends a `FINISH_SENDER` frame on a stream. It contains the number of messages ever sent reliably on this channel. All `SENT_UNRELIABLE` frames for this channel must have been sent on this same stream. After doing this, the endpoint may no longer send any more frames pertaining to this channel, except `FORGET_CHANNEL` frames.

When an application cancels a channel via a sender handle, the endpoint sends a `CANCEL_SENDER` frame on a stream. After doing this, the endpoint may no longer finish this channel. Upon doing this, the endpoint should reset all streams on which it is sending `MESSAGE` frames for that channel.

### 3.4 § Acking and nacking

Shortly after an endpoint receives a `MESSAGE` frame on a stream, it must send an `ACK_RELIABLE` frame acking it. This contains ranges of reliable message numbers the endpoint has received. The sequence of varints in RANGES alternates between the length of a gap before the next range, followed by the length of the next range. The first gap is relative to zero. Gaps merely represent ranges that this frame is not acking, rather than nacks.

An endpoint must maintain an "unreliable receipt deadline" of how long to wait to receive an unreliable message before declaring it lost. This may be set to 1 second or twice the current estimated RTT. When an endpoint receives a `SENT_UNRELIABLE` frame it learns that the remote endpoint has sent all unreliable MESSAGE_NUMs less than the sum of all received SENT_UNRELIABLE.COUNT, for that channel.

Shortly after the unreliable receipt deadline after learning that the remote side has sent a new unreliable MESSAGE_NUM the endpoint must ack it if it has received it and nack it if it hasn't. This is done by sending an `ACK_NACK_UNRELIABLE` frame on a stream. The sequence of varints in RANGES alternates between the length of the next range to ack and the length of the next range to nack. The first range is relative to where the last `SENT_UNRELIABLE` frame for the channel left off.

All `SENT_UNRELIABLE` frames for a channel must be sent on the same stream.

When an endpoint receives a `MESSAGE` frame from an unreliable datagram for which it has already nacked its unreliable MESSAGE_NUM, it must ignore it. The exception to this is that the endpoint does not have to uphold this behavior in cases when its receiver state machine for the channel was destroyed then re-created since sending the nack.

### 3.5 § Channel state machine lifecycle

When an endpoint receives a `ROUTE_TO` frame for which CHANNEL.CREATOR is remote and state for it does not exist, it creates a sender/receiver state machine for it. Upon doing so it must also ensure it sends back a `ROUTE_TO` frame for the channel back to the remote shortly after (this avoids certain lost in transit detection race conditions).

When an endpoint receives a `ROUTE_TO` frame for which CHANNEL.CREATOR is local and state for it does not exist, it sends a `FORGET_CHANNEL` frame on a stream for that channel and ignores the rest of the frame sequence.

When an endpoint receives a `FORGET_CHANNEL` frame it must destroy its sender/receiver state machine for the channel if one exists, and should abort any ongoing operations to send frames for the channel, and should reset any streams being used to send frames for the channel. This rule supersedes any other requirements that an endpoint do things when certain circumstances occur.

When an endpoint sends a `MESSAGE` frame it creates a sender/receiver state machine for each attachment. When an endpoint receives a `MESSAGE` frame it creates sender/receiver state machines for each attachment if they do not yet exist.

When an application closes a channel's receiver it must send a `CLOSE_RECEIVER` frame on a stream. All `ACK_RELIABLE` and `ACK_NACK_UNRELIABLE` frames for the channel must have been sent on the same stream. The `CLOSE_RECEIVER` frame constitutes a nack for all reliable and unreliable MESSAGE_NUMs of the channel not already acked or nacked. After sending this, the endpoint must destroy its receiver state machine for the channel, but not abandon ongoing operations to send frames for the channel.

When an endpoint receives a `CANCEL_SENDER` frame for a channel, it must do the same thing it does when the application closes the channel's receiver.

When an endpoint receives a `FINISH_SENDER` frame for a channel, it must wait to receive all messages sent reliably on the channel, and also wait for all messages sent unreliably on the channel to be acked or nacked. It must not nack packets more eagerly than it would otherwise because the channel is finishing. Upon these conditions being met, it must do the same thing it does when the application closes the channel's receiver. This process gets preempted if the receiver state machine is destroyed for some other reason.

An endpoint should track the set of channel state machines for which CHANNEL.CREATOR is remote that have been destroyed recently (e.g. within 1 second). Messages routed to such a channel should be ignored. This rule supersedes any other requirements that an endpoint do things when certain circumstances occur.

An endpoint should track the set of channels for which it has sent a `FORGET_CHANNEL` frame recently (e.g. within 1 second). Any requirements that a `FORGET_CHANNEL` frame be sent for a channel due to some trigger can and should be waived if the endpoint already such a frame more recently than the trigger. An endpoint that implements this logic must take care to use a monotonic clock for this logic, whether logical or physical.

### 3.6 § Channel lost in transit detection

We define distributed facts about the connection that a networked channel may be "accessible" or "lost". At a given point in time a channel may be neither accessible nor lost, but once a channel becomes accessible or lost it stays as such forever. A channel cannot be both accessible and lost. It may be the case that a channel is accessible or lost but either side of the connection does not yet know that this is the case.

We define the entrypoint channel as always being accessible. A channel is accessible if the message it was attached to was acked and also sent on a channel that is accessible. A channel is lost if the message it was attached to was nacked or if it was sent on a channel that is lost.

If an endpoint does not yet know that a channel is accessible or lost, it considers it to be "pending".

When an endpoint learns that a channel it created is lost, it must destroy its state machine for the channel and send a `FORGET_CHANNEL` frame on a stream for that channel. The exception to this is that it is not required to send the remote side the `FORGET_CHANNEL` frame if it has never sent the remote side a `ROUTE_TO` frame for the channel. It should also abandon any ongoing operations to send frames for the channel and reset any streams being used for that, except for the operation and stream to send a `FORGET_CHANNEL` stream. This rule supersedes any other requirements that an endpoint do things when certain circumstances occur.

**3.6.1 § Analysis of required state** This section describes what state and mechanism is required to implement channel lost in transit detection. It contains no information that couldn't be logically deduced from other sections.

First off, it is worth noting that any channel for which the local application has dequeued a handle is accessible. The fact that the message it was attached to was dequeued means that that message was acked. If the channel that message was sent on was created by the local side, it is either the entrypoint channel or a channel that the remote side dequeued a sender for (we know this because it sent a message on it). If the channel that message was sent on was created by the remote side, it is either the entrypoint channel or a channel that the local side dequeued a receiver for (we know this because it dequeued a message from it). In any of these cases, the channel that message was sent on is accessible (proof by induction). Therefore, the channel for which the local application has dequeued a handle is also accessible.

An endpoint can maintain a boolean variable for each sender state machine to track whether it is marked as accessible, as opposed to pending. It is not necessary to represent the lost state, and a sender state machine is immediately destroyed when it's known that it's lost. The entrypoint channel and any channel created by the remote side start as accessible. Any channel created by the local side other than the entrypoint channel start as pending.

An endpoint can also maintain for each sender state machine a multimap from each un-acked message number it sent on that sender to all additional channels attached to that message. Finally, for only sender state machines which are still in a pending state, the endpoint can maintain a collection of all additional channels attached to acked messages sent on that sender.

Whenever it first becomes the case that a sender is marked as accessible and some message sent on it with attachments has been acked, the endpoint can mark all its attached channels as accessible. For attached senders, this may set off a recursive cascade of marking channels as accessible. For attached receivers, no work is necessary.

Conversely, whenever a message with attachments is nacked, the endpoint deals with the fact that it now knows that all attached channels are lost. For each attached channel, it can send `FORGET_CHANNEL` frames if needed and destroy those channels' state. If those lost channels are

senders, the endpoint learns that all other channels attached to messages sent to the lost channels are also lost, even if the messages they were attached to were acked. This may set off a recursive cascade of learning that channels have been lost and dealing with that.

### 3.7 § Flow control

Each side