This is a non-normative document intended to give an intuitive understanding of the Aqueduct protocol's architecture.

An Aqueduct connection runs over a QUIC connection, between the client and the server. Other than a few exceptions, the protocol is symmetrical between client and server.

Aqueduct is primarily concerned with maintaining corresponding state machines for linked sender/receiver pairs on opposite sides of the connection. An Aqueduct connection ultimately is a distributed system, albeit one that ontologically only has 2 nodes. Typically within a distributed system a lot of the complexity comes from the fact that messages may not be processed by the thing processing them in a predictable order relative to each other due to them being processed by entirely different nodes, and also because it may be possible for some messages to be lost entirely without the entire system fail-stopping. However, due to the power of the QUIC protocol, we have engineered ways to bring these problems down to the humble 2-node system as well!

It is conventional to for an Aqueduct implementation to allow the user to construct a channel in an initially non-networked state, and then make the channel networked by sending one side of it within a message on another networked channel. However, this is purely an API nicety, and not something the protocol itself is aware of. From the perspective of the protocol implementation, the connection begins with solely the entrypoint channel existing, and additional channels are created by sending a message on a pre-existing channel with the newly created channel attached to it. This naturally leads to the channels within a connection forming a tree structure of which channel was used to create another channel, wherein the root of the tree is the entrypoint channel, and all other channels have a unique lineage of parent channels linking them back to the entrypoint. It is notable that the nodes of this tree can flip back and forth in terms of which side of the connection created them and which direction the channel's messages flow—however, a channel's creator-side will always equal the sender-side of its parent channel.

Channels within an Aqueduct connection are uniquely identified by channel IDs. A channel ID is a bit field consisting of 3 boolean flags and a 61-bit sequential integer ID. The 3 boolean flags indicate whether the channel was created by the client or by the server, whether the channel is flowing to the server or to the client, and whether the channel is multishot or oneshot. The sequential integer ID is allocated sequentially by a counter unique to the combination of the other 3 bits and owned by the side of the connection that is creating the channel. The channel ID consisting entirely of zeroes is known as the entrypoint channel, and is treated specially in certain ways.

Over the wire, the Aqueduct protocol is based on the two sides sending frames to each other over both QUIC unidirectional streams and QUIC unreliable datagrams. The most central frame type is the MESSAGE frame, which the sender side of a channel sends to convey a message being sent on that channel. A MESSAGE frame morally contains the channel ID it is being sent on, the message payload (a byte string), and a list of additional channel IDs that are attached to the message (corresponding to a collection of additional channels that get newly created by the sending of this message).

- If a sender is operating in ORDERED mode, all of its MESSAGE frames are sent on the same QUIC stream.
- If a sender is operating in UNORDERED mode, each of its MESSAGE frames are sent on different QUIC streams.
- If a sender is operating in UNRELIABLE mode, each of its MESSAGE frames are sent on QUIC unreliable datagrams, unless they are too big to fit in an unreliable datagram, in which case Aqueduct falls back to sending it in a QUIC stream.

MESSAGE frames are morally the only frame type that Aqueduct sometimes sends in unreliable

datagrams. There are a variety of other frame types dedicated to maintaining and the lifecycle of channels, the connection, and other important things; and these are only sent on QUIC streams.

This document will guide the reader to an intuitive understanding of the design of the Aqueduct protocol in the following way:

- First, we will imagine how the Aqueduct protocol could be designed if we assumed all frames would be delivered reliably and in the correct order (basically, if we were constructing Aqueduct on top of a TCP connection), and also if we assumed that neither side of the connection ever forgets information (so, we allow both sides' memory consumption to grow unboundedly).
- Next, we will introduce relaxations to our assumptions regarding ordering and reliability, and discuss how the Aqueduct protocol handles the various possible race conditions and other complications caused by that.
- Finally, we will introduce rules for when the Aqueduct implementation is allowed to delete certain state and reclaim its memory, discuss what possible race conditions and complications that can cause (especially in conjunction with message reordering and loss), and discuss how the Aqueduct protocol handles that as well.

Imagine first the simplest case of a client which creates the connection, sends some messages to the server on the entrypoint channel, and then finishes the channel:

frames client sends to server
msg
msg
msg
finish

The server would simply enqueue these messages into a buffer for the server-side application to dequeue. Once the server receives the frame indicating that the channel is finished, it similarly allows its application to observe this fact.

There are basically 3 ways the Aqueduct API can be used to intentionally close a channel:

- By the sender-side finishing it.
- By the sender-side cancelling it.
- By the receiver-side dropping the receiver.

If the client closed this channel by cancelling it rather than by finishing it, this simplified protocol execution could be quite similar. Rather than sending a FINISH frame, the client could send a CANCEL frame. The main difference would be that upon the server receving this frame it would drop any of the messages it enqueued in its buffer that the server-side application hadn't yet dequeued:

frames client sends to server
msg
msg
msg
cancel

In the case that the server-side application closed the channel by dropping its receiver, the server could drop any enqueued messages and send back a frame to the client telling it that the receiver

has been dropped. Upon receiving this, the client would allow its application to observe this fact:

frames client sends to server	frames server sends to client
msg	drop receiver
msg	
msg	

It's worth noting that, in both such cases, the client doesn't inherently know how many of the messages it sent the server-side application dequeued before dropping the rest and entering a state of ignoring additional ones. Moreover, in the receiver-dropping case, the client sending message frames and even possibly even a finish or cancel frame can occur in parallel to the server sending a drop receiver frame, such that both occur before their transmitting side has received the frames from the other.

Imagine, then, that the client creates an additional client-to-server channel by attaching it to one of the entrypoint channel messages, and sends messages on the second channel as well:

frames client sends to server
chan 1 msg
chan 1 msg
chan 1 msg (attachments=[chan 2])
chan 2 msg
chan 2 msg
chan 1 msg
chan 1 finish
chan 2 msg
chan 2 finish

When the server receives chan 1 msg 2, it knows to create a receiver state machine for chan 2, since it was attached. After that point, the server can process messages pertaining to chan 2. In this simplified protocol, the relative ordering between frames pertaining to chan 1 and chan 2 doesn't matter except that messages pertaining to chan 2 must occur after the chan 1 frame which creates chan 2.