The Aqueduct Protocol: Design Overview

Aqueduct is a network protocol based on the central idea of sending channels within channels, built on top of QUIC. This document introduces the design and motivations of Aqueduct and describes the overall structure of how an application can use it.

Channels

The most important object that an Aqueduct implementation provides is a "channel". A channel is a queue of messages with sender and receiver handles that can be used simultaneously by multiple parallel tasks to coordinate. This should be a concept that is familiar to many engineers.

A channel can be local, meaning that the senders and receivers are in the same process. Alternatively, a channel can be networked, such that messages sent by the sender half are encoded and transmitted over an Aqueduct connection to the process with the receiver half. Code built over Aqueduct can be largely oblivious to whether a given channel is local or networked.

Creating channels

The above notions are useful, but not particularly novel–the key feature of Aqueduct lies in how a networked channel is created. Almost all channels are initially created as local channels, by calling a constructor that returns a linked sender/receiver pair. Then, by sending the sender half or the receiver half down a different pre-existing networked channel, the new channel automatically and seamlessly becomes networked, within the same Aqueduct connection as the pre-existing channel.

This allows code built over Aqueduct to be oblivious not only to whether it is using networked or local channels, but also to whether it is *creating* networked or local channels. This property allows application developers to create powerful, asynchronous, and network-optimized abstractions that cross-cut network boundaries.

The one channel not created in this way is the "entrypoint" channel–the first networked channel constructed by a client connecting to a server. In this case, the server creates a receiver handle by binding to a socket address, and a client creates a sender handle by connecting to its socket address. This creates a new Aqueduct connection, and its entrypoint channel from which all other networked channels are bootstrapped.

Back pressure

The sender side of a channel can unilaterally decide whether the channel experiences back pressure or not. This is opaque to the receiver. The burden of buffering messages lies on the sender-side process.

Receiver closing

If the receiver side of a channel closes its receiver handle(s), the sender side is notified of this, and it manifests as an error upon the sender trying to send.

Sender finishing

One of the operations the sender side of a channel can do is to "finish" the channel. This represents a graceful closure of the message stream. Once the receiver side has dequeued all messages sent before the stream was finished, further attempts to dequeue a message yield a value indicating that the channel has finished. This value is not considered an error.

Sender cancelling

Another operation the sender side of a channel can do is to "cancel" the channel. This represents an un-graceful aborting of the message stream and an as-soon-as-possible abandonment of all enqueued messages. Existing enqueued messages on both sides are dropped, and subsequent attempts by the receiver to dequeue a message yield an error.

Aqueduct uses QUIC's "stream reset" functionality to implement sender cancelling in a highly efficient way.

Reliability mode

The sender side of a channel can unilaterally decide what "reliability mode" a channel operates in, between these three options:

1. ORDERED: All messages sent are received in the order they are sent.

   Aqueduct implements ORDERED mode by sending all messages for the channel in the same QUIC stream. This is the simplest mode, suitable for most purposes.

2. UNORDERED: All messages sent are received, but may be reordered.

   Aqueduct implements UNORDERED mode by sending each message for the channel in a different QUIC stream. This improves head of line blocking; If a UDP datagram containing (part of) a message is dropped, subsequent messages in the channel can be delivered immediately rather than waiting for the dropped message to be re-transmitted.

3. UNRELIABLE: Not all sent messages are guaranteed to be received.

   Aqueduct implements UNRELIABLE mode by sending each message for the channel in a QUIC unreliable datagram. This is very similar to just sending the message in a plain old UDP datagram (albeit with encryption), and is suitable for highly latency-sensitive operations. If a UDP datagram containing (part of) a message is dropped, the message is lost forever.

Note: Claims that "all messages sent are received" are still subject to exceptions such as the entire connection being lost, the sender cancelling, or the receiver dropping.

Note: Even if all channels in an application use ORDERED mode, Aqueduct still helps the application avoid head of line blocking between different channels, as different channels still use different QUIC streams.

Channel lost in transit errors

There are several ways a message sent on a channel can be lost in transit, despite the connection as a whole remaining alive: a message sent in UNRELIABLE mode may be lost randomly, or the sender could cancel the channel before the message is dequeued, or the receiver could be closed before the message is dequeued. An important edge case can occur here if the message that was lost included sender or receiver handles to additional channels: even if the process that created the channel still has a sender or receiver handle to it, that channel is useless because the remote process will never dequeue its corresponding receiver or sender handle. In this case, the remote application is not even capable of performing proper cleanup logic.

The Aqueduct protocol contains a mechanism to help applications handle this edge case elegantly. Any message that is permanently lost in transit will trigger any channels attached to the message to also be considered lost in transit. Subsequent attempts to use that channel's sender or receiver will yield a "channel lost in transit" error. Moreover, any messages enqueued into the channel before that point will automatically be marked as lost in transit, which may potentially trigger a multi-level cascade of other channels and messages being marked as lost in transit; Aqueduct handles this case efficiently and elegantly. This mechanism vastly reduces the set of edge cases an application may have to handle in certain situations of partial failure.

Constraints on channel topology

The Aqueduct protocol does impose certain constraints on what network topologies channels can have:

1. A channel cannot have multiple sender handles in multiple different processes, nor multiple receiver handles in multiple different processes.
2. Only a sender or receiver handle created locally can be sent to a different process. Handles cannot bunny-hop between processes.
3. A process cannot send both the sender and receiver handles to a channel to different processes. At least one side must remain in its original process.
4. A channel can only have a sender or receiver handle attached to a message once. Different network-crossing handles connect to different channels.

These constraints significantly simplify the set of edge cases an application has to handle, especially in contexts with malicious processes. This is because these constraints make it possible to statically reason about what connection each channel will be associated with.

Oneshot channels

Aqueduct is primarily concerned with multishot channels, wherein multiple messages can be sent one after another. However, Aqueduct also provides oneshot channels, wherein only a single message can ever be sent. These are useful for various things, such as simulating a remote procedure call architecture.

Metadata headers

An Aqueduct implementation allows the application to install certain kinds of middleware capable of reading and setting HTTP-like headers (metadata key/value pairs) on Aqueduct connections, channels, and individual messages. Metadata headers aren't meant to be overused, but they do have certain use cases. For example, a connection-level metadata header is used to make sure the client and server agree on the serialization format of application messages. Metadata headers may also be useful for telemetry purposes, such as to maintain logical clocks for debugging info.

Other ambient benefits of QUIC

In addition the the above mentioned benefits, Aqueduct also passively benefits from other features of QUIC, including always-on TLS encryption, simultaneous transport and cryptographic handshake to reduce connection start-up time, the ability of a connection to seamlessly and automatically survive a change in client IP address, and a design that is conducive to efficient stateless packet-level load balancers and reverse proxies.

Future work

In the future, we are interested in exploring how to create Aqueduct-style abstractions that let applications effectively leverage QUIC's 0-RTT data functionality, and also NAT traversal / peer-to-peer hole punching. We are also interested in any users attempting to use Aqueduct to create audio/video streaming systems–if you've noticed any ways that Aqueduct creates frictions for these use cases, please reach out to us.

We are also tentatively interested in creating abstractions for the sender or receiver handle to an already-networked channel to be seamlessly handed off to a different process. However, we only want to do this if we can find a way to avoid it compromising the edge case simplifications Aqueduct's design and constraints otherwise achieves.

See also

The Aqueduct protocol specification: `PROTOCOL.md`.