QMux

draft-opik-quic-qmux

IETF 124 Montreal

Kazuho Oku*, Lucas Pardue, Jana Iyengar, Eric Kinnear

the updated charter

The fourth area of work is the specification of how QUIC stream multiplexing and other application-oriented extensions (e.g. Datagram) can be adapted to work over a reliable and bidirectional byte stream substrate. When the substrate is insecure, TLS will be the default security provider; no effort will be made to enable unprotected communication without a security provider. Substrates must provide congestion-management capabilities applicable to their deployment environments.

recall the comparison at IETF 123

4 approaches in a nutshell

	frame encoding	negotiation	establishment latency*	record layer
a) WT-over-H1	capsules	HTTP header	3 RT	TLS
b) capsules	capsules	QUIC v1 TP?	2 RT	TLS
c) v1 frames	QUIC v1	QUIC v1 TP	2 RT	TLS
d) v1 frames & packets	QUIC v1	QUIC v1 TP	2 RT	QUIC v1-ish
(QUIC v1)	QUIC v1	QUIC v1 TP	1 RT	QUIC v1

* measured as the time between when the client sends the first packet of the connection (e.g., TCP SYN) and when it sends the first flight of application data, while doing a full handshake

complexity depends on perspective

	complexity*
coding from scratch**	b ~ c < a < d
coding on top of QUIC v1 stacks***	c ~ d < b < a
coding on top of WT-over-H2 stacks****	a < b < c < d
specifying the protocol ("uncertainties")	a < c < b < d

a) WT-over-H1, b) capsules, c) v1 frames, d) v1 frames & packets

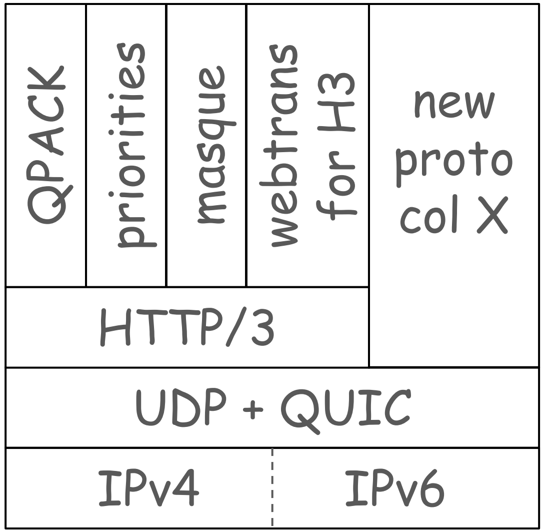
*: the leftmost one is the least complex
**: *a* needs HTTP; *d* requires implementing the packetization layer (incl. AEAD)
***: all require buffering of incomplete capsules, frames, or packets; *b* uses different frame encoding and tightens the stream state machinery; *a* requires use of HTTP in addition
****: *a* and *b* are derivations of WT-over-H2, *c* and *d* are not; *d* requires implementing packetization

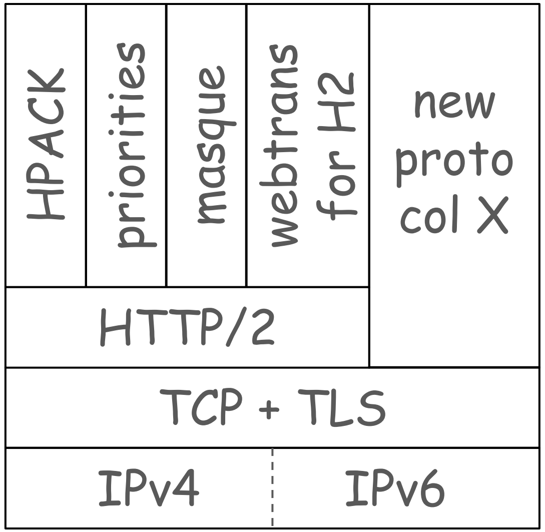
draft-opik-quic-qmux

formerly known as QUIC-on-Streams
the v1 frames approach

sad state for applications

different application protocols built on top of QUIC and TCP



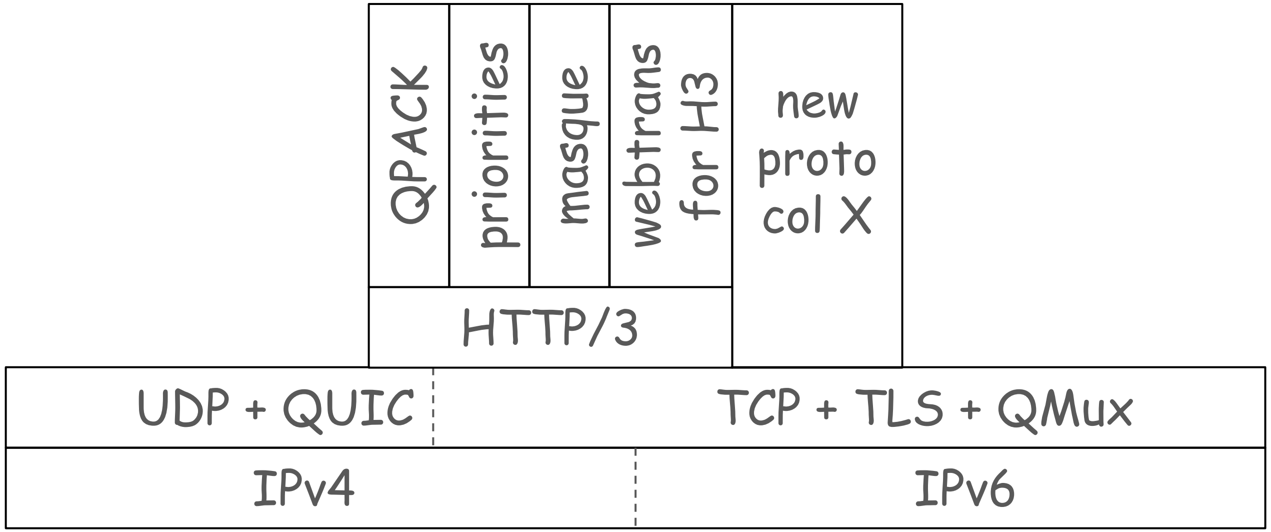


we cannot stop using TCP

- reasons:
  - UDP port blocking
  - QUIC's unnecessary overhead when used on transports that provide reliability and/or protection (e.g., UNIX socket, RoCEv2 RC)

need for a unified interface

Backport the QUIC API contract (i.e., QUIC streams + datagrams) to TCP, so that QUIC applications can be run on top of TCP too.



need for a unified interface

Backport the QUIC API contract (i.e., QUIC streams + datagrams) to TCP, so that QUIC applications can be run on top of TCP too.

Maybe a bit too late for some application protocols, but better late than never.

the goal of QMux

- provide a unified interface to protocol designers and application developers
- the smaller the divergence is from QUIC v1, the easier it becomes for QUIC stacks to add support for QMux
- doing so maximizes the chance of application developers seeing their QUIC stack of choice support QMux

the goal of QMux (cont'd)

- retain QUIC v1 as much as possible, rather than trying to be clever
  - stream state machinery
  - frame encodings
- outsource the heavyweight functions to substrates
  - loss detection and congestion control
  - packetization and encryption

QMux: protocol overview

- negotiation
- frame encoding / decoding
- no acks
- permitted and prohibited frames
- implementation experience and strategy

negotiation

- relies on TLS ALPN for application protocol negotiation
- first frame being sent by each endpoint is the QX_TRANSPORT_PARAMETERS frame; carries TPs
    - bans the use of TPs unrelated to data transfer
    - max_frame_size TP communicates the frame size of STREAM / DATAGRAM frames that omit the length field
        - default is 16KB (i.e., the default max. of a TLS record)

frame encoding

- same as QUIC v1
- for each stream, STREAM frames are always sent in order
  - no need for buffering or reassembly
  - i.e., the offset field of STREAM frames is a pure redundancy

decoding frames

- endpoints need to be prepared for receiving partial frames, as the underlying substrate is byte-oriented
- what the receiver could do:
  - when receiving new data, call the QUIC v1 frame decoder
    - if the frame is truncated, retain the input (as opposed to generating errors in QUIC v1)
    - when more bytes are received, repeat this process of speculative decoding, until a complete frame is received
- i.e., no need for a dedicated logic to handle frame truncation

decoding frames (cont'd)

- <u>as an optimization</u>, receivers might stream the payload of STREAM frames to the application before a complete STREAM frame is received
  - there's little reason to do so when TLS is used (and max_frame_size is set to the default 16KB), because if a STREAM frame fits into a single TLS record, the complete STREAM frame would always become available at once

no ACKs

- ACK frames aren't used; they are prohibited
    - because retransmission is the task of the substrate
- implementation advice:
    - when adding support for QMux, QUIC v1 stacks can pretend that frames are ACKed the moment they are written to the TCP socket

permitted and prohibited frames

- permitted:
  - PADDING, PING, RESET_STREAM, STOP_SENDING, MAX_*, *_BLOCKED, CONNECTION_CLOSE (application-level only), DATAGRAM, RESET_STREAM_AT
- prohibited:
  - ACK, CRYPTO, NEW_TOKEN, NEW_CONNECTION_ID, RETIRE_CONNECTION_ID, PATH_CHALLENGE, PATH_RESPONSE, HANDSHAKE_DONE

implementation experience

- working PoC for quicly created in 1/2 day
- took another 1/2 day to integrate that into h2o and speak H3-over-QMux
- working PoC for quic-go created and achieved interop with quicly/h2o during the hackathon

implementation strategy (quicly)

- sending side:
  - allocate buffer for building a QUIC packet, but set the capacity to 16KB and skip generating the packet header
  - build frames like you do with QUIC v1
    - doing so registers the ACK callbacks
  - run all the ACK callbacks
  - pass the buffer to the substrate

implementation strategy (quicly)

- changes to QUIC v1 code:
    - frame decoders return PARTIAL error when the given frame image is truncated
    - change the QUIC v1 receive function, so that when a frame decoder returns a PARTIAL error, it is converted to a FRAME_ENCODING error
- when receiving QMux:
    - repeatedly call the frame decoders until PARTIAL error is returned
    - retain the remaining bytes until more bytes are received

open questions

Q1. Is the draft heading in the right direction?

Q2. Send frames (capsules) directly on bidirectional byte streams?

The alternative: send packets.

Q3. Minimize divergence from QUIC v1?

The alternative is to optimize, e.g.:

a. omit the offset field of STREAM frames
b. tighten stream state machinery to prevent use of out-of-date / redundant frames

Q4. Frame encoding: QUIC v1 or capsules?

Q1. is the draft heading in the right direction?

our take:

Yes. The draft is simple and meets the updated charter; i.e.:

- supports stream multiplexing and transfer of datagrams
- works on reliable, bidirectional byte stream substrate
- TLS is the default security provider

Q2. send frames (capsules) directly?

our take:

Yes. Packetization and encryption should be offloaded to the substrate, rather than done inside QMux. This is because the substrates are often optimized for the purpose (e.g., hardware offload) and we want to leverage them.

Q3. minimize divergence from QUIC v1?

our take:

Yes. Diverging adds cost to standardization, implementations, and testing. Added cost reduces the likelihood of QUIC stacks supporting QMux, which in turn increases pain to application developers.

Q4. Frame encoding: QUIC v1 or capsules?

our take:

Use QUIC v1.

- There are far more QUIC stacks than WT/H2.
- We want to see existing QUIC stacks support QMux to minimize the pain of application developers.
- Easier to manage frame encoding and registry inside QUIC WG.

open questions

Q1. Is the draft heading in the right direction?

Q2. Send frames (capsules) directly on bidirectional byte streams?

The alternative: send QUIC packets.

Q3. Minimize divergence from QUIC v1?

The alternative is to optimize, e.g.:

a. omit the offset field of STREAM frames
b. tighten stream state machinery to prevent use of out-of-date / redundant frames

Q4. Frame encoding: QUIC v1 or capsules?