Applying, Observing, and Debugging QUIC

Lucas Pardue
QUIC is not TCP
QUIC is *not* TLS
QUIC is not HTTP
QUIC is **not** the web over UDP
QUIC is QUIC
QUIC is a secure transport protocol
QUIC is what you make it
Ain’t got the time?

It all starts with a handshake.

Then, application data can be sent using reliable streams or unreliable datagrams.

QUIC packets are protected. If you don’t have the keys, you can’t see contents.

Reliable data is retransmitted in new packets. Packets are *not* retransmitted.
Applicability and Management

Want to send your application data over QUIC? Read RFC 9308 - “Applicability of the QUIC Transport Protocol”

Operate a network and want to observe/manage QUIC? Read RFC 9312 - “Manageability of the QUIC Transport Protocol”
Everything starts with a handshake

- RFC 9000, Section 7 - Cryptographic and Transport Handshake
- RFC 9001 - Using TLS to Secure QUIC
- RFC 9312, Section 2.4 - The QUIC Handshake
- The specs detail it all
  - Jana and MT walked us through during Monday’s session
- Key items: Initial and Handshake packets
  - Initial is a type, not an adjective
  - Easy to misinterpret “Initial packet” as “initial (first) packet” - that way leads to sadness
The illustrated guide

Sometimes it helps to look at things differently than the specs.

https://quic.xargs.org/
(source code: https://github.com/syncsynchalt/illustrated-quic)
The session begins with the client sending an "Init" packet. This packet contains the "Client-hello" TLS record, used to begin the TLS 1.3 encrypted session.

---

The encrypted session begins with the client saying "Hello". The client provides information including the following:

- client random data (used later in the handshake)
- a list of cipher suites that the client supports
- a public key for key exchange
- protocol versions that the client can support

---

Any data sent by the client that contains an Initial packet must be padded to a length of 1200 bytes. This library does it by appending null bytes to the data stream.

---

Padding

Padding this packet to a size of 1200 bytes serves two purposes:

- **Path MTU detection**: Any IPv4 host or router is allowed to drop packets that exceed their MTU limit, to a minimum of 576 bytes. The vast majority of the internet has a much higher MTU (typically 1500 bytes). A higher packet size will increase throughput and performance. Given these realities QUIC chooses a minimum size constraint of 1200 bytes, which should traverse the vast majority of real networks (including bottleneck networks) without being dropped for size.

To prevent a scenario where a connection is established successfully with smaller packets but then starts dropping once larger packets are sent, the initial packets are padded to a length of 1200 bytes to prove that the end-to-end path will allow packets of that size.

- **Amplification Attack Mitigation**: There is a class of network attack in which an attacker can send a small amount of traffic to an innocent third-party which replies with a much larger amount of traffic directed at the target. In the case of QUIC this could be done with IP address spoofing, and would cause QUIC servers to reply to small initial datagrams with much larger Handshake responses. To help mitigate this, QUIC servers are forbidden from replying to a client with more than 3 times the traffic that was sent to it, until the server has received some proof from the client that it’s at the given address (such as round-trip data originally from the server). Adding padding to this initial datagram gives the server a "byte budget" to perform Handshake responses without exceeding this 3x limit.
Transport Parameters

Remember: QUIC Transport Parameters are a TLS extension

https://www.iana.org/assignment/s/quic/quic.xhtml
Illustration on live connections

Our old friends pcap and Wireshark.

To successfully dissect QUIC packets, Wireshark 3.4.x and onwards. Examples use Cloudflare quiche - https://github.com/cloudflare/quiche.


Server: quiche-server --no-retry
Ready-made examples

Follow along examples at https://github.com/LPardue/ietf-115-tdd

“localhost-good”
Client Initial - ALPN

**RFC 7301** - Application-Layer Protocol Negotiation

Client offers a list of all the application protocols it would like to speak over this connection.

https://www.iana.org/assignments/tls-extensiontype-values/tls-extensiontype-values.xhtml#alpn-protocol-ids
Server Initial and Handshake

Secrets not available!
Keys needed to see the full picture

From even a very early stage in a connection, QUIC packets are encrypted with a session key.

SSLKEYLOGFILE is an approach used by many, but not all, implementations. Endpoints can be instructed to explicitly log their keys to the nominated file. draft-thomson-tls-keylogfile I-D seeking to formalise the format.

Session keys are symmetrical, either endpoint can log to enable packet decryption in both directions.

Server Initial and Handshake (w/ keys)

https://wiki.wireshark.org/TLS

Server selects one ALPN, this is the application protocol that will be used over QUIC.

Applications need to agree on how QUIC streams are used.

Now we can dissect QUIC
Live illustration revisited

Dissection without keys

<table>
<thead>
<tr>
<th>No.</th>
<th>Time</th>
<th>Source</th>
<th>Src port</th>
<th>Destination</th>
<th>Dst port</th>
<th>Protocol</th>
<th>Length Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.00690000</td>
<td>127.0.0.1</td>
<td>43959</td>
<td>127.0.0.1</td>
<td>4433</td>
<td>QUIC</td>
<td>Initial, DCID=6c4d2c29cbbcf6525a262cb22c9cb42, SID=0463b0d665a7b2d189da2871fc25977b7c67b, PKN: 0, CRYPTO</td>
</tr>
<tr>
<td>2</td>
<td>0.00172373</td>
<td>127.0.0.1</td>
<td>43959</td>
<td>127.0.0.1</td>
<td>4439</td>
<td>QUIC</td>
<td>Handshake, DCID=54c63b0d665a7b2d189da2871fc25977b7c67b, PKN: 0, CRYPTO</td>
</tr>
<tr>
<td>3</td>
<td>0.003943505</td>
<td>127.0.0.1</td>
<td>43959</td>
<td>127.0.0.1</td>
<td>4433</td>
<td>QUIC</td>
<td>Handshake, DCID=7061de665a7b2d189da2871fc25977b7c67b, PKN: 0, CRYPTO</td>
</tr>
<tr>
<td>4</td>
<td>0.00213033</td>
<td>127.0.0.1</td>
<td>43959</td>
<td>127.0.0.1</td>
<td>4433</td>
<td>QUIC</td>
<td>Handshake, DCID=643b0d665a7b2d189da2871fc25977b7c67b, PKN: 0, CRYPTO</td>
</tr>
<tr>
<td>5</td>
<td>0.004963775</td>
<td>127.0.0.1</td>
<td>43959</td>
<td>127.0.0.1</td>
<td>4433</td>
<td>QUIC</td>
<td>Handshake, DCID=7061de665a7b2d189da2871fc25977b7c67b, PKN: 0, CRYPTO</td>
</tr>
<tr>
<td>6</td>
<td>0.005933516</td>
<td>127.0.0.1</td>
<td>43959</td>
<td>127.0.0.1</td>
<td>4433</td>
<td>QUIC</td>
<td>Handshake, DCID=7061de665a7b2d189da2871fc25977b7c67b, PKN: 0, CRYPTO</td>
</tr>
<tr>
<td>7</td>
<td>0.005253457</td>
<td>127.0.0.1</td>
<td>43959</td>
<td>127.0.0.1</td>
<td>4433</td>
<td>QUIC</td>
<td>Handshake, DCID=643b0d665a7b2d189da2871fc25977b7c67b, PKN: 0, CRYPTO</td>
</tr>
<tr>
<td>8</td>
<td>0.006139349</td>
<td>127.0.0.1</td>
<td>43959</td>
<td>127.0.0.1</td>
<td>4433</td>
<td>QUIC</td>
<td>Handshake, DCID=7061de665a7b2d189da2871fc25977b7c67b, PKN: 0, CRYPTO</td>
</tr>
<tr>
<td>9</td>
<td>0.006153643</td>
<td>127.0.0.1</td>
<td>43959</td>
<td>127.0.0.1</td>
<td>4433</td>
<td>QUIC</td>
<td>Handshake, DCID=7061de665a7b2d189da2871fc25977b7c67b, PKN: 0, CRYPTO</td>
</tr>
<tr>
<td>10</td>
<td>0.007994888</td>
<td>127.0.0.1</td>
<td>43959</td>
<td>127.0.0.1</td>
<td>4439</td>
<td>QUIC</td>
<td>Handshake, DCID=54c63b0d665a7b2d189da2871fc25977b7c67b, PKN: 0, CRYPTO</td>
</tr>
<tr>
<td>11</td>
<td>0.007235874</td>
<td>127.0.0.1</td>
<td>43959</td>
<td>127.0.0.1</td>
<td>4433</td>
<td>QUIC</td>
<td>Handshake, DCID=7061de665a7b2d189da2871fc25977b7c67b, PKN: 0, CRYPTO</td>
</tr>
<tr>
<td>12</td>
<td>0.007425998</td>
<td>127.0.0.1</td>
<td>43959</td>
<td>127.0.0.1</td>
<td>4433</td>
<td>QUIC</td>
<td>Handshake, DCID=7061de665a7b2d189da2871fc25977b7c67b, PKN: 0, CRYPTO</td>
</tr>
<tr>
<td>13</td>
<td>0.007554547</td>
<td>127.0.0.1</td>
<td>43959</td>
<td>127.0.0.1</td>
<td>4433</td>
<td>QUIC</td>
<td>Handshake, DCID=7061de665a7b2d189da2871fc25977b7c67b, PKN: 0, CRYPTO</td>
</tr>
<tr>
<td>14</td>
<td>0.007779237</td>
<td>127.0.0.1</td>
<td>43959</td>
<td>127.0.0.1</td>
<td>4433</td>
<td>QUIC</td>
<td>Handshake, DCID=7061de665a7b2d189da2871fc25977b7c67b, PKN: 0, CRYPTO</td>
</tr>
<tr>
<td>15</td>
<td>0.007979760</td>
<td>127.0.0.1</td>
<td>43959</td>
<td>127.0.0.1</td>
<td>4433</td>
<td>QUIC</td>
<td>Handshake, DCID=7061de665a7b2d189da2871fc25977b7c67b, PKN: 0, CRYPTO</td>
</tr>
<tr>
<td>16</td>
<td>0.008855673</td>
<td>127.0.0.1</td>
<td>43959</td>
<td>127.0.0.1</td>
<td>4433</td>
<td>QUIC</td>
<td>Handshake, DCID=7061de665a7b2d189da2871fc25977b7c67b, PKN: 0, CRYPTO</td>
</tr>
<tr>
<td>17</td>
<td>0.008487620</td>
<td>127.0.0.1</td>
<td>43959</td>
<td>127.0.0.1</td>
<td>4433</td>
<td>QUIC</td>
<td>Handshake, DCID=7061de665a7b2d189da2871fc25977b7c67b, PKN: 0, CRYPTO</td>
</tr>
</tbody>
</table>
CIDs

Whether the packets are encrypted or not, connection IDs are visible. And they can be used for traffic steering / load balancing, as described by Ian and Martin.

<table>
<thead>
<tr>
<th>Client</th>
<th>Server DCID:</th>
<th>78015def011d1adf3af94c44067955dd4d52fc70</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server</td>
<td>Client DCID:</td>
<td>9463b9d6695a7b2d189da2871fc255977bc7c6f8</td>
</tr>
</tbody>
</table>

Client -> Server DCID: 78015def011d1adf3af94c44067955dd4d52fc70
Server-> Client DCID: 9463b9d6695a7b2d189da2871fc255977bc7c6f8
qlog - structured logging by endpoints

Implementations often have debugging that can enhance or augment packet captures.

A common logging format can encourage an ecosystem of analysis tools. E.g. what is an endpoint producing and why is it doing that?


Draft-ietf-quic-qlog-quic-events, draft-ietf-quic-qlog-h3-events: concrete definitions to cover events related to packets and frames, security, congestion control etc.
Making sense out of oodles of data

https://qvis.quictools.info/
A real-world failure in wireshark -
“localhost-0-streams-uni”

Client: SSLKEYLOGFILE=tdd.keys QLOGDIR=qlogs quiche-client --no-verify --wire-version 1 --max-streams-uni 0 https://127.0.0.1:4433/index.html

### Frame 10: 189 bytes on wire (1464 bits), 183 bytes captured (1464 bits) on interface lo, id 0
- **Ethernet II, Src: 00:00:00:00:00:00 (00:00:00:00:00:00), Dst: 00:00:00:00:00:00 (00:00:00:00:00:00)**
- **Internet Protocol Version 5, Src: 127.0.0.1, Dst: 127.0.0.1**
- **User Datagram Protocol, Src Port: 4439, Dst Port: 56795**
- **QUIC IETF**
- **QUIC IETF**

**QUIC Short Header**
- **DCID: 2665e60d5b6e407f152fe8f69b818e47b4**; **PKN=0**
- **Header Form: Short Header (0)**
- **Fixed Bit: True**
- **Spin Bit: False**
- **Reserved: 0**
- **Key Phase Bit: False**
- **Packet Number Length: 1 bytes (0)**
- **Destination Connection ID: 2665e60d56e407b7527c56626569b91e48a724**
- **Packet Number: 0**

**Protected Payload: Aa55e6d492752b88f107f9b94c6f3c2aafdf415ab2421c756e619934bad2cbb898c....**

**CONNECTION_CLOSE (Application)**
- **Error code**: 0xff
- **Reason phrase length**: 20

“Error opening control stream”
The same real-world failure in qvis

CONNECTION_CLOSE

“Error opening control stream”
Another real-world failure

"lucaspardue.com-0-streams-uni"

Client: SSLKEYLOGFILE=tdd.keys QLOGDIR=qlogs quiche-client
--wire-version 1 --max-streams-uni 0 https://lucaspardue.com/index.html

Where’s the connection CONNECTION_CLOSE?
Another real-world failure (2)

The thing that sticks out is the trace is longer and there is no CONNECTION_CLOSE received by the server.

Where’s the connection CONNECTION_CLOSE?
Debugging the difference

1) Attempting to open a connection with initial_max_streams_uni = 0 to localhost elicits a CONNECTION_CLOSE from the server.

2) Attempting to open a connection with initial_max_streams_uni = 0 to lucaspardue.com causes no packets to be returned.

3) Different implementation? Not really, both servers powered by the same QUIC library.

So what could be the root cause?
Different types of CONNECTION_CLOSE

RFC 9000, Section 10.2.3

CONNECTION_CLOSE frame of type 0x1c is for transport layer.

CONNECTION_CLOSE frame of type 0x1d is for application layer.

“Sending a CONNECTION_CLOSE of type 0x1d in an Initial or Handshake packet could expose application state or be used to alter application state. A CONNECTION_CLOSE of type 0x1d MUST be replaced by a CONNECTION_CLOSE of type 0x1c when sending the frame in Initial or Handshake packets. Otherwise, information about the application state might be revealed. Endpoints MUST clear the value of the Reason Phrase field and SHOULD use the APPLICATION_ERROR code when converting to a CONNECTION_CLOSE of type 0x1c.”
Trouble with timing, causing timeouts

- Server, uses an HTTP/3 library.
- It sees initial_max_streams_uni = 0, it calls the QUIC library close() function, passing an error code and reason.
- Neither application nor HTTP/3 library check the transport state before closing it.
- Timing differences speaking to lucaspardue.com over the Internet.
- Handshake not complete when an application (0x1d) CONNECTION_CLOSE was triggered.
- Unsafe to send application errors => the server would not send a packet.
- After close(), server no longer processes client packets.
- Client retires, but eventually idle time out kicks in and it gives up.
Debugging leads to a fix

https://github.com/cloudflare/quiche/pull/1355

Automatically check the transport layer connection state and choose the most appropriate and safe type of error to emit.

Client always receives a timely close.
Summary

QUIC is not TCP, TLS, HTTP nor “the web over UDP”

QUIC is QUIC. It provides transport services for applications, such as multiplexed reliable byte streams. It doesn’t have much opinion about how these get used; see RFC 9308 for guidance and considerations for application protocols on top of QUIC. Define an ALPN identifier!

Minimal information in the wire image is observable; see RFC 9312. QUIC packets used in the handshake use a deterministic key. Once a secure connection is established, unique session keys are used.

Implementations and deployments can behave differently. Techniques that can decrypt (SSLKEYLOGFILE) or log plain text (qlog) can help analysis or debug.
Backup slides
TransportPacketSent = {
    header: PacketHeader

    ? frames: [* $QuicFrame]
    ? is_coalesced: bool .default false
    ? retry_token: Token
    ? stateless_reset_token: StatelessResetToken

    ? supported_versions: [+ QuicVersion]

    ? raw: RawInfo
    ? datagram_id: uint32

    ? is_mtu_probe_packet: bool .default false

    ? trigger:
        "retransmit_reordered" /
        "retransmit_timeout" /
        "pto_probe" /
        "retransmit_crypto" /
        "cc_bandwidth_probe"
}

; The QuicFrame is any key-value map (e.g., JSON object)
$QuicFrame /= {
    * text => any
}

$QuicFrame /= QuicBaseFrames

QuicBaseFrames /=
    PaddingFrame / PingFrame / AckFrame / ResetStreamFrame /
    StopSendingFrame / CryptoFrame / NewTokenFrame / StreamFrame /
    MaxDataFrame / MaxStreamDataframe / MaxStreamsFrame /
    DataBlockedFrame / StreamDataBlockedFrame / StreamsBlockedFrame /
    NewConnectionIDFrame / RetireConnectionIDFrame /
    PathChallengeFrame / PathResponseFrame / ConnectionCloseFrame /
    HandshakeDoneFrame / UnknownFrame

PaddingFrame = {
    frame_type: "padding"

    ; total frame length, including frame header
    ? length: uint32
    payload_length: uint32
}

...
qlog example

Client: QLOGDIR=qlogs quiche-client --no-verify --wire-version 1
https://127.0.0.1:4433/index.html

Server: QLOGDIR=qlogs quiche-server --no-retry

{"qlog_version": "0.3", "qlog_format": "JSON-SEQ", "title": "quiche-client qlog", "description": "quiche-client qlog id=9463b9d6695a7b2d189da2871fc255977bc7c6f8", "trace": {"vantage_point": {"type": "client"}, "title": "quiche-client qlog", "description": "quiche-client qlog id=9463b9d6695a7b2d189da2871fc255977bc7c6f8", "configuration": {"time_offset": 0.0}}}

{"time": 0.0, "name": "transport:parameters_set", "data": {"owner": "local", "tls_cipher": "None", "disable_active_migration": true, "max_idle_timeout": 30000, "max_udp_payload_size": 1350, "ack_delay_exponent": 3, "max_ack_delay": 25, "active_connection_id_limit": 2, "initial_max_data": 10000000, "initial_max_stream_data_id": 10000000, "initial_max_stream_data bidi": 100, "initial_max_stream_data_id": 10000000, "initial_max_stream_data bidi": 100, "initial_max_streams_id": 100, "initial_max_streams bidi": 100, "initial_max_streams bidi": 100}}

{"time": 0.207949, "name": "transport:packet_sent", "data": {"header": {"packet_type": "initial", "packet_number": 0, "version": "1", "scil": 20, "dcil": 16, "scid": "9463b9d6695a7b2d189da2871fc255977bc7c6f8", "dcid": "6c94d2c299cf66253a202bc20ce942"}, "raw": {"length": 350, "payload_length": 287}, "send_at_time": 0.207949, "frames": [{"frame_type": "crypto", "offset": 0, "length": 283}]}}

{"time": 0.207949, "name": "recovery:metrics_updated", "data": {"smoothed_rtt": 333.0, "rtt_variance": 166.5, "congestion_window": 13500, "bytes_in_flight": 350, "ssthresh": 18446744073709551615}}

{"time": 3.5715451, "name": "transport:packet_received", "data": {"header": {"packet_type": "initial", "packet_number": 0, "version": "1", "scil": 20, "dcil": 20, "scid": "78015def011d1adf3af94c44067955dd4d52fc70", "dcid": "9463b9d6695a7b2d189da2871fc255977bc7c6f8"}, "raw": {"length": 1200, "payload_length": 117}, "frames": [{"frame_type": "ack", "ack_delay": 0.305, "acked_ranges": [0, 0]}]}

{"frame_type": "crypto", "offset": 0, "length": 90}\

QUIC Deep Dive, IETF 115, Tuesday 8 November 2022
Congestion control behavior
Applicability of QUIC - streams

Streams are a core capability of RFC 9000.

Streams in QUIC provide a lightweight, *ordered* byte-stream abstraction to an application.

Streams can be created by either endpoint, can concurrently send data interleaved with other streams, and can be canceled. QUIC **does not provide any means of ensuring ordering between bytes on different streams**.

QUIC allows for an arbitrary number of streams to operate concurrently and for an arbitrary amount of data to be sent on any stream, subject to flow control constraints and stream limits.
Applicability of QUIC - stream IDs

- Streams can be unidirectional or bidirectional.
- Unidirectional streams carry data in one direction: from the initiator of the stream to its peer.
- Bidirectional streams allow for data to be sent in both directions. Streams are identified within a connection by a numeric value, referred to as the stream ID.
- A stream ID is a 62-bit integer (0 to $2^{62}-1$) that is unique for all streams on a connection.
- The least significant bit (0x01) of the stream ID identifies the initiator of the stream. The second least significant bit (0x02) of the stream ID distinguishes between bidirectional and unidirectional.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Stream Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Client-Initiated, Bidirectional</td>
</tr>
<tr>
<td>0x01</td>
<td>Server-Initiated, Bidirectional</td>
</tr>
<tr>
<td>0x02</td>
<td>Client-Initiated, Unidirectional</td>
</tr>
<tr>
<td>0x03</td>
<td>Server-Initiated, Unidirectional</td>
</tr>
</tbody>
</table>
What does all that mean?

Applications have a toolkit of streams to use.

QUIC has no opinion how you use those streams, as long as the transport requirements on IDs and flow control are obeyed.

Application mappings like HTTP/3 (RFC 9114) or DNS over QUIC (RFC 9250) describe how application messages utilise QUIC streams.
Streams example: HTTP/3

Client-initiated bidirectional streams are always used for request and response exchanges.

Client- and server-initiated unidirectional streams have a type, conveyed in the first byte(s) of the stream.

Each endpoint creates mandatory unidirectional control streams: Control, QPACK encoder, QPACK decoder.

HTTP/3 defines its own framing layer on top of QUIC. HTTP/3 frames are sent on QUIC streams.
Streams example: HTTP/3

Control stream on ID 2. QPACK streams on ID 6 and 10.

Request stream on ID 0. GET request for /index.html. Stream is FIN’d to indicate request message is complete.
Streams example: HTTP/3
Stream gotchas for applications 1

Concurrency and flow control have limits.

An endpoint tells its peer the initial limits using Transport Parameters in the QUIC handshake.

QUIC control frames like MAX_STREAMS, MAX_DATA, MAX_STREAM_DATA can be used to update limits during the connection lifetime.

QUIC doesn’t have an opinion. This is an application matter. There is no universal default. Implementations of applications probably have an opinion on defaults and behaviours.
Stream gotchas for applications 2

Transport Parameters apply to a QUIC connection, they affect applications.

Clients can offer many types of application protocols in their ALPN.

Servers can only pick one.

Applications might have specifications that disagree on suitable Transport Parameters.

For example, HTTP/3 control streams are mandatory. If an endpoint never gives credit to its peer to allow these streams to be opened, the peer might get upset.
Stream multiplexing

Unlike TCP, QUIC offers multiplexing of byte streams within the connection. This offers fruitful capability and fertile ground for new behaviours that might be hard to observe or debug.

Streams compete for connection bandwidth. Not all streams are equal. E.g., streams for a control channel might be more important than bulk data.

QUIC does not provide global ordering of stream data in a connection. Stream IDs indicate stream creation order but data from different streams can arrive at any time. Applications that depend on ordering across streams need to implement application-layer synchronization.
Example: HTTP/3 prioritization shown in qvis

5 concurrent transfers of 5 MB, all urgency=1

quiche (before priorities)
  round-robin

quiche (now)
  FIFO