QUIC

A UDP-Based Multiplexed and Secure Transport


Simon Pietro Romano -- spromano@unina.it
The big picture

- QUIC is a multiplexed and secure general-purpose transport protocol providing:
  - Stream multiplexing
  - Stream and connection-level flow control
  - Low-latency connection establishment
  - Connection migration and resilience to NAT rebinding
  - Authenticated and encrypted header and payload
Streams, connections, packets, frames

- Streams
  - the basic service abstraction that QUIC provides
- Connections
  - the context in which QUIC endpoints communicate
- Packets and Frames
  - the basic unit used by QUIC to communicate
Streams

- A lightweight, ordered byte-stream abstraction to an application
- Unidirectional or bidirectional
- An alternative view of QUIC unidirectional streams:
  - a "message" abstraction of practically unlimited length
- Can be created by either endpoint, can concurrently send data interleaved with other streams, and can be cancelled
- 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
Stream types

- Streams are identified within a connection by a numeric value, referred to as the stream ID
  - a 62-bit integer (0 to $2^{62}-1$) that is unique for all streams on a connection
  - encoded as variable-length integers
- The least significant bit of the stream ID identifies the initiator of the stream
  - client-initiated streams have even-numbered stream IDs
  - server-initiated streams have odd-numbered stream IDs
- The second least significant bit of the stream ID distinguishes between bidirectional streams (with the bit set to 0) and unidirectional streams (with the bit set to 1)
- The least significant two bits from a stream ID therefore identify a stream as one of four types
<table>
<thead>
<tr>
<th>Bits</th>
<th>Stream Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>Client-Initiated, Bidirectional</td>
</tr>
<tr>
<td>0x1</td>
<td>Server-Initiated, Bidirectional</td>
</tr>
<tr>
<td>0x2</td>
<td>Client-Initiated, Unidirectional</td>
</tr>
<tr>
<td>0x3</td>
<td>Server-Initiated, Unidirectional</td>
</tr>
</tbody>
</table>

Table 1: Stream ID Types
Sending and receiving data

- STREAM frames encapsulate data sent by an application
- An endpoint uses the Stream ID and Offset fields in STREAM frames to place data in order
- Endpoints must be able to deliver stream data to an application as an ordered byte-stream
- Delivering an ordered byte-stream requires that an endpoint buffer any data that is received out of order, up to the advertised flow control limit
- The data at a given offset must not change if it is sent multiple times
  - an endpoint may treat receipt of different data at the same offset within a stream as a connection error of type PROTOCOL_VIOLATION
- An endpoint MUST NOT send data on any stream without ensuring that it is within the flow control limits set by its peer
Prioritizing streams

- A QUIC implementation should provide ways in which an application can indicate the relative priority of streams.
- When deciding which streams to dedicate resources to, the implementation should use the information provided by the application.
Sending and receiving streams

- **Sending side:**
  - write data (if flow control credit exists)
  - end the stream, via a STREAM frame with the FIN bit set
  - reset the stream, via a RESET_STREAM frame

- **Receiving side:**
  - read data
  - abort reading of the stream and request closure, possibly resulting in a STOP_SENDING frame
Sending part of a stream

- In the "Send" state, an endpoint transmits (and retransmits as necessary) stream data in STREAM frames.
- The endpoint respects the flow control limits set by its peer, and continues to accept and process MAX_STREAM_DATA frames.
- An endpoint in the "Send" state generates STREAM_DATA_BLOCKED frames if it is blocked from sending by stream or connection flow control limits.
Receiving part of a stream

- In the "Recv" state, the endpoint receives STREAM and STREAM_DATA_BLOCKED frames.
- Incoming data is buffered and can be reassembled into the correct order for delivery to the application.
- As data is consumed by the application and buffer space becomes available, the endpoint sends MAX_STREAM_DATA frames to allow the peer to send more data.
Flow control

- It is necessary to limit the amount of data that a receiver could buffer
  - prevent a fast sender from overwhelming a slow receiver
  - prevent a malicious sender from consuming a large amount of memory at a receiver.
- Streams are flow controlled both individually and as an aggregate
- A QUIC receiver controls the maximum amount of data the sender can send on a stream at any time
- Similarly, to limit concurrency within a connection, a QUIC endpoint controls the maximum cumulative number of streams that its peer can initiate
- An endpoint limits the cumulative number of incoming streams a peer can open
Data flow control

- QUIC employs a credit-based flow-control scheme, where a receiver advertises the number of bytes it is prepared to receive on a given stream and for the entire connection.
- This leads to two levels of data flow control in QUIC:
  - Stream flow control, which prevents a single stream from consuming the entire receive buffer for a connection by limiting the amount of data that can be sent on any stream.
  - Connection flow control, which prevents senders from exceeding a receiver's buffer capacity for the connection, by limiting the total bytes of stream data sent in STREAM frames on all streams.
- A receiver
  - sets initial credits for all streams by sending transport parameters during the handshake.
  - sends MAX_STREAM_DATA or MAX_DATA frames to the sender to advertise additional credit.
QUIC connections

- QUIC's connection establishment combines version negotiation with the cryptographic and transport handshakes to reduce connection establishment latency
- Once established, a connection may migrate to a different IP or port at either endpoint
- A connection may be terminated by either endpoint
Connection IDs

- Each connection possesses a set of connection identifiers, each of which can identify the connection.
- Connection IDs ensure that changes in addressing at lower protocol layers (UDP, IP) don't cause packets for a QUIC connection to be delivered to the wrong endpoint.
- Connection IDs must not contain any information that can be used by an external observer to correlate them with other connection IDs for the same connection.
- Packets with long headers include Source Connection ID and Destination Connection ID fields.
  - These fields are used to set the connection IDs for new connections.
- Packets with short headers only include the Destination Connection ID and omit the explicit length (which is expected to be known to endpoints.
Issuing connection IDs

- The initial connection ID issued by an endpoint is sent in the Source Connection ID field of the long packet header during the handshake.
- Additional connection IDs are communicated to the peer using NEW_CONNECTION_ID frames.
- An endpoint can change the connection ID it uses for a peer to another available one at any time during the connection.
- An endpoint consumes connection IDs in response to a migrating peer.
- When the endpoint wishes to remove a connection ID from use, it sends a RETIRE_CONNECTION_ID frame to its peer.
  - Sending a RETIRE_CONNECTION_ID frame indicates that the connection ID will not be used again and requests that the peer replace it with a new connection ID using a NEW_CONNECTION_ID frame.
Matching packets to connections

- Incoming packets are classified on receipt
- Packets can either be associated with an existing connection, or (for servers) potentially create a new connection
- Hosts try to associate a packet with an existing connection
  - If the packet has a non-zero-length Destination Connection ID corresponding to an existing connection, QUIC processes that packet accordingly
  - If the Destination Connection ID is zero length and the packet matches the local address and port of a connection where the host used zero-length connection IDs, QUIC processes the packet as part of that connection
Version negotiation

- Version negotiation ensures that client and server agree to a QUIC version that is mutually supported
- A server sends a Version Negotiation packet in response to each packet that might initiate a new connection
- The size of the first packet sent by a client will determine whether a server sends a Version Negotiation packet
  - Clients that support multiple QUIC versions should pad the first packet they send to the largest of the minimum packet sizes across all versions they support
  - This ensures that the server responds if there is a mutually supported version
Version negotiation: client vs server behavior

- If the version selected by the client is not acceptable to the server, the server responds with a Version Negotiation packet.
- When a client receives a Version Negotiation packet, it must abandon the current connection attempt.
- Once the Version Negotiation packet is determined to be valid, the client selects an acceptable protocol version from the list provided by the server.
- The client then attempts to create a new connection using that version.
- The new connection must use a new random Destination Connection ID different from the one it had previously sent.
Cryptographic and transport handshake

- QUIC relies on a combined cryptographic and transport handshake to minimize connection establishment latency
- QUIC uses the CRYPTO frame to transmit the cryptographic handshake
- Version 0x00000001 of QUIC uses TLS
- QUIC provides reliable, ordered delivery of the cryptographic handshake data
- QUIC packet protection is used to encrypt as much of the handshake protocol as possible
Cryptographic handshake properties

● Authenticated key exchange, where:
  ○ a server is always authenticated
  ○ a client is optionally authenticated
  ○ every connection produces distinct and unrelated keys
  ○ keying material is usable for packet protection for both 0-RTT and 1-RTT packets
  ○ 1-RTT keys have forward secrecy

● Authenticated values for transport parameters of both endpoints, and confidentiality protection for server transport parameters

● Authenticated negotiation of an application protocol
  ○ e.g., Application-Layer Protocol Negotiation Extension (ALPN) for TLS
1-RTT handshake

Client

Initial[0]: CRYPTO[CH] ->

Server

Initial[0]: CRYPTO[SH] ACK[0]
Handshake[0]: CRYPTO[EE, CERT, CV, FIN]
  <- 1-RTT[0]: STREAM[1, "..."]

Initial[1]: ACK[0]
Handshake[0]: CRYPTO[FIN], ACK[0]
1-RTT[0]: STREAM[0, "..."], ACK[0] ->

1-RTT[1]: STREAM[3, "..."], ACK[0]
  <- Handshake[1]: ACK[0]
1-RTT packets and frames

- The cryptographic handshake is used to agree on cryptographic keys
- It is carried in “Initial” and “Handshake” packets
- Each line shows a QUIC packet with the packet type and packet number shown first, followed by the frames that are typically contained in those packets
  - So, for instance the first packet is of type Initial, with packet number 0, and contains a CRYPTO frame carrying the ClientHello
- Multiple QUIC packets may be coalesced into a single UDP datagram
  - so the handshake may consist of as few as 4 UDP datagrams
0-RTT handshake

Client                      Server

Initial[0]: CRYPTO[CH]      Initial[0]: CRYPTO[SH] ACK[0]
0-RTT[0]: STREAM[0, "..."] ->
Handshake[0] CRYPTO[EE, FIN]
<- 1-RTT[0]: STREAM[1, "..."] ACK[0]

Initial[1]: ACK[0]
Handshake[0]: CRYPTO[FIN], ACK[0]
1-RTT[1]: STREAM[0, "..."] ACK[0] ->

1-RTT[1]: STREAM[3, "..."], ACK[1]
<- Handshake[1]: ACK[0]
QUIC address validation

- Used by QUIC to avoid being used for a traffic amplification attack
- Primary defense against amplification attack is verifying that an endpoint is able to receive packets at the transport address that it claims
- Address validation is performed both during connection establishment and during connection migration
- Upon receiving the client's Initial packet, the server can, e.g., request address validation by sending a Retry packet containing a token
  - this token must be repeated by the client in all Initial packets it sends for that connection after it receives the Retry packet
Adding a Retry message to the handshake

Client

Initial[0]: CRYPTO[CH]  \to\  

Initial+Token[1]: CRYPTO[CH]  \to\  Initial[0]: CRYPTO[SH]  ACK[1]  

Handshake[0]: CRYPTO[EE, CERT, CV, FIN]  \leftarrow  1-RTT[0]: STREAM[1, "..."]
Path validation

- Used during connection migration by the migrating endpoint to verify reachability of a peer from a new local address
- Endpoints test reachability between a specific local address and a specific peer address
  - an address is the two-tuple of IP address and port
- Path validation tests that packets (PATH_CHALLENGE) can be both sent to and received (PATH_RESPONSE) from a peer on the path
- It also validates that the packets received from the migrating endpoint do not carry a spoofed source address
Connection migration

- The use of a connection ID allows connections to survive changes to endpoint addresses (IP address and port), such as those caused by an endpoint migrating to a new network.
- An endpoint may probe for peer reachability from a new local address using path validation prior to migrating the connection to the new local address.
- An endpoint uses a new connection ID for probes sent from a new local address.
- An endpoint can migrate a connection to a new local address by sending packets containing non-probing frames from that address.
- Receiving acknowledgments for data sent on the new path serves as proof of the peer's reachability from the new address.
Connection termination

- An established QUIC connection can be terminated in one of three ways:
  - idle timeout
  - immediate close
  - stateless reset
- An endpoint may discard connection state if it does not have a validated path on which it can send packets
- The “closing” and “draining” connection states exist to ensure that connections close cleanly and that delayed or reordered packets are properly discarded.
  - These states should persist for at least three times the current value of the ad hoc defined Probe Timeout (PTO) interval
Stateless reset

- A stateless reset is provided as an option of last resort for an endpoint that does not have access to the state of a connection.
- An endpoint may send a stateless reset in response to receiving a packet that it cannot associate with an active connection.

Figure 6: Stateless Reset Packet
Packets and frames

- QUIC endpoints communicate by exchanging packets
- Packets have confidentiality and integrity protection and are carried in UDP datagrams
- QUIC uses the long packet header during connection establishment
- Packets with the long header are:
  - Initial, 0-RTT, Handshake, Retry
  - Version negotiation uses a version-independent packet with a long header
- Packets with the short header are designed for minimal overhead and are used after a connection is established and 1-RTT keys are available
Long header packets

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>T</th>
<th>T</th>
<th>X</th>
<th>X</th>
<th>X</th>
<th>X</th>
</tr>
</thead>
</table>

Packet type

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>Initial</td>
</tr>
<tr>
<td>0x1</td>
<td>0-RTT</td>
</tr>
<tr>
<td>0x2</td>
<td>Handshake</td>
</tr>
<tr>
<td>0x3</td>
<td>Retry</td>
</tr>
</tbody>
</table>

Version (32)

DCID Len (8)

Destination Connection ID (0..160) ...

SCID Len (8)

Source Connection ID (0..160) ...
Short header packets

| 0 | 1 | S | R | R | K | P | P |
+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |
| Destination Connection ID (0..160) | ... |
|   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |
| Packet Number (8/16/24/32) | ... |
|   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |
| Protected Payload (*) | ... |
|   |   |   |   |   |   |   |   |
QUIC frames

- The payload of QUIC packets, after removing packet protection, consists of a sequence of complete frames

- Version Negotiation, Stateless Reset, and Retry packets do not contain frames
Figure 8: Generic Frame Layout
## Frame types

<table>
<thead>
<tr>
<th>Type Value</th>
<th>Frame Type Name</th>
<th>Value Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>PADDING</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x01</td>
<td>PING</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x02 - 0x03</td>
<td>ACK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x04</td>
<td>RESET_STREAM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x05</td>
<td>STOP_SENDING</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x06</td>
<td>CRYPTO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x07</td>
<td>NEW_TOKEN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08 - 0x0f</td>
<td>STREAM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td>MAX_DATA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x11</td>
<td>MAX_STREAM_DATA</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x12 - 0x13</td>
<td>MAX_STREAMS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x14</td>
<td>DATA_BLOCKED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x15</td>
<td>STREAM_DATA_BLOCKED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x16 - 0x17</td>
<td>STREAMS_BLOCKED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x18</td>
<td>NEW_CONNECTION_ID</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x19</td>
<td>RETIRE_CONNECTION_ID</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x1a</td>
<td>PATH_CHALLENGE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x1b</td>
<td>PATH_RESPONSE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x1c - 0x1d</td>
<td>CONNECTION_CLOSE</td>
</tr>
</tbody>
</table>
Getting familiar with QUIC

- https://datatracker.ietf.org/wg/quic/charter/
  - QUIC charter @ IETF
- https://www.youtube.com/watch?v=ZF6-KmHz1sU
  - QUIC tutorial @IETF98 (Chicago, March 2017)
- https://github.com/quicwg/base-drafts
  - a bunch of useful resources
  - a list of ongoing implementation efforts
An example library: LiteSpeed QUIC

**Isquic**

LiteSpeed QUIC and HTTP/3 library. Works on Linux, FreeBSD, MacOS, and Windows. Turn-key open-source web server that uses Isquic is available at [openlitespeed.org](http://openlitespeed.org) in both source and package forms.

- **Language:** C
- **Version:** Draft-24, Draft-23, Q039, Q043, and Q046.
- **Roles:** Client, Server, Library
- **Handshake:** QUIC Crypto, RFC 8446
- **Protocol IDs:** 0xff000018, 0xff000017
- **Public server:**
  - http3-test.litespeedtech.com:4433, http3-test.litespeedtech.com:4434 (sends stateless retry packets), and http3-test.litespeedtech.com:4435 (faster downloads due to less logging) for ID-24 and ID-23 as well as Google QUIC versions Q039, Q043, and Q046
    - This server supports HTTP/3 and QPACK and provides some services to test transfer of data each way. `GET /` for details.
lsquic: build and run with Docker

Building with Docker

The library and the example client and server can be built with Docker.

Initialize Git submodules:

```bash
cd lsquic
git submodule init
git submodule update
```

Build the Docker image:

```bash
docker build -t lsquic .
```

Then you can use the examples from the command line. For example:

```bash
sudo docker run -it --rm lsquic http_client --s www.google.com -p / -o version=0046
sudo docker run -p 12345:12345/udp -v /path/to/certs:/mnt/certs -it --rm lsquic http_server -c www.example
```