# Introduction

QUIC (Quick UDP Internet Connection) is a new multiplexed and secure transport atop UDP, designed from the ground up and optimized for HTTP/2 semantics. While built with HTTP/2 as the primary application protocol, QUIC builds on decades of transport and security experience, and implements mechanisms that make it attractive as a modern general-purpose transport. QUIC provides multiplexing and flow control equivalent to HTTP/2, security equivalent to TLS, and connection semantics, reliability, and congestion control equivalent to TCP.

QUIC operates entirely in userspace, and is currently shipped to users as a part of the Chromium browser, enabling rapid deployment and experimentation. As a userspace transport atop UDP, QUIC allows innovations which have proven difficult to deploy with existing protocols as they are hampered by legacy clients and middleboxes, or by prolonged Operating System development and deployment cycles.

An important goal for QUIC is to inform better transport design through rapid experimentation. As a result, we hope to inform and where possible migrate distilled changes into TCP and TLS, which tend to have much longer iteration cycles.

This document describes the conceptual design and the wire specification of the QUIC protocol prior to standardization. Accompanying documents describe the combined crypto and transport handshake [QUIC-CRYPTO], and loss recovery and congestion control [draft-iyengar-quic-loss-recovery]. Additional resources, including a more detailed rationale document, are available on the Chromium QUIC webpage.

Proposals for standardization of QUIC based on this early deployment are [draft-hamilton-quic-transport-protocol], [draft-shade-quic-http2-mapping], [draft-iyengar-quic-loss-recovery], and [draft-thomson-quic-tls].

# Conventions and Definitions

All integer values used in QUIC, including length, version, and type, are in little-endian byte order, and not in network byte order. QUIC does not enforce alignment of types in dynamically sized frames.

A few terms that are used throughout this document are defined below.

- "Client": The endpoint initiating a QUIC connection.
- "Server": The endpoint accepting incoming QUIC connections.
- "Endpoint": The client or server end of a connection.
- "Stream": A bi-directional flow of bytes across a logical channel within a QUIC connection.
- "Connection": A conversation between two QUIC endpoints with a single encryption context that multiplexes streams within it.
- "Connection ID": The identifier for a QUIC connection.
- "QUIC Packet": A well-formed UDP payload that can be parsed by a QUIC receiver. QUIC packet size in this document refers to the UDP payload size.

# A QUIC Overview

We now briefly describe QUIC's key mechanisms and benefits. QUIC is functionally equivalent to TCP+TLS+HTTP/2, but implemented on top of UDP. Key advantages of QUIC over TCP+TLS+HTTP/2 include:

- Connection establishment latency
- Flexible congestion control
- Multiplexing without head-of-line blocking
- Authenticated and encrypted header and payload
- Stream and connection flow control
- Connection migration

### Connection Establishment Latency

QUIC combines the crypto and transport handshakes, reducing the number of roundtrips required for setting up a secure connection. QUIC connections are commonly 0-RTT, meaning that on most QUIC connections, data can be sent immediately without waiting for a reply from the server, as compared to the 1-3 roundtrips required for TCP+TLS before application data can be sent.

QUIC provides a dedicated stream (Stream ID 1) to be used for performing the handshake, but the details of this handshake protocol are out of this document's scope. For a complete description of the current handshake protocol, please see the QUIC Crypto Handshake document. QUIC current handshake will be replaced by TLS 1.3 in the future.

### Flexible Congestion Control

QUIC has pluggable congestion control and richer signaling than TCP, which enables QUIC to provide richer information to congestion control algorithms than TCP. Currently, the default congestion control is a reimplementation of TCP Cubic; we are currently experimenting with alternative approaches.

One example of richer information is that each packet, both original and retransmitted, carries a new packet sequence number. This allows a QUIC sender to distinguish ACKs for retransmissions from ACKs for original transmissions, thus avoiding TCP's retransmission ambiguity problem. QUIC ACKs also explicitly carry the delay between the receipt of a packet and its acknowledgment being sent, and together with the monotonically-increasing packet numbers, this allows for precise roundtrip-time (RTT) calculation.

Finally, QUIC's ACK frames support up to 256 ack blocks, so QUIC is more resilient to reordering than TCP (with SACK), as well as able to keep more bytes on the wire when there is reordering or loss. Both client and server have a more accurate picture of which packets the peer has received.

### Stream and Connection Flow Control

QUIC implements stream- and connection-level flow control, closely following HTTP/2's flow control. QUIC's stream-level flow control works as follows. A QUIC receiver advertises the absolute byte offset within each stream upto which the receiver is willing to receive data. As data is sent, received, and delivered on a particular stream, the receiver sends WINDOW_UPDATE frames that increase the

advertised offset limit for that stream, allowing the peer to send more data on that stream.

In addition to per-stream flow control, QUIC implements connection-level flow control to limit the aggregate buffer that a QUIC receiver is willing to allocate to a connection. Connection flow control works in the same way as stream flow control, but the bytes delivered and highest received offset are all aggregates across all streams.

Similar to TCP's receive-window autotuning, QUIC implements autotuning of flow control credits for both stream and connection flow controllers. QUIC's autotuning increases the size of the credits sent per WINDOW_UPDATE frame if it appears to be limiting the sender's rate, and throttles the sender when the receiving application is slow.

### Multiplexing

HTTP/2 on TCP suffers from head-of-line blocking in TCP. Since HTTP/2 multiplexes many streams atop TCP's single-bytestream abstraction, a loss of a TCP segment results in blocking of all subsequent segments until a retransmission arrives, irrespective of the HTTP/2 stream that is encapsulated in subsequent segments.

Because QUIC is designed from the ground up for multiplexed operation, lost packets carrying data for an individual stream generally only impact that specific stream. Each stream frame can be immediately dispatched to that stream on arrival, so streams without loss can continue to be reassembled and make forward progress in the application.

Caveat: QUIC currently compresses HTTP headers via HTTP/2 HPACK header compression on a dedicated header stream(3), which imposes head-of-line blocking for header frames only.

### Authenticated and Encrypted Header and Payload

TCP headers appear in plaintext on the wire and not authenticated, causing a plethora of injection and header manipulation issues for TCP, such as receive-window manipulation and sequence-number overwriting. While some of these are active attacks, others are mechanisms used by middleboxes in the network sometimes in an attempt to transparently improve TCP performance. However, even "performance-enhancing" middleboxes still effectively limit the evolvability of the transport protocol, as has been observed in the design of MPTCP and in its subsequent deployability issues.

QUIC packets are always authenticated and typically the payload is fully encrypted. The parts of the packet header which are not encrypted are still authenticated by the receiver, so as to thwart any packet injection or manipulation by third parties. QUIC protects connections from witting or unwitting middlebox manipulation of end-to-end communication.

Caveat: PUBLIC_RESET packets that reset a connection are currently not authenticated.

### Connection Migration

TCP connections are identified by a 4-tuple of source address, source port, destination address and destination port. A well-known problem with TCP is that connections do not survive IP address changes (for example, by switching from WiFi to cellular) or port number changes (when a client's

NAT binding expires causing a change in the port number seen at the server). While MPTCP addresses the connection migration problem for TCP, it is still plagued by lack of middlebox support and lack of OS deployment.

QUIC connections are identified by a 64-bit Connection ID, randomly generated by the client. QUIC can survive IP address changes and NAT re-bindings since the Connection ID remains the same across these migrations. QUIC also provides automatic cryptographic verification of a migrating client, since a migrating client continues to use the same session key for encrypting and decrypting packets.

In cases when the connection is unambiguously identified by the 4-tuple, such as when a server sends packets to a client using an ephemeral port, there is an option to not send the connection ID to save bytes on the wire.

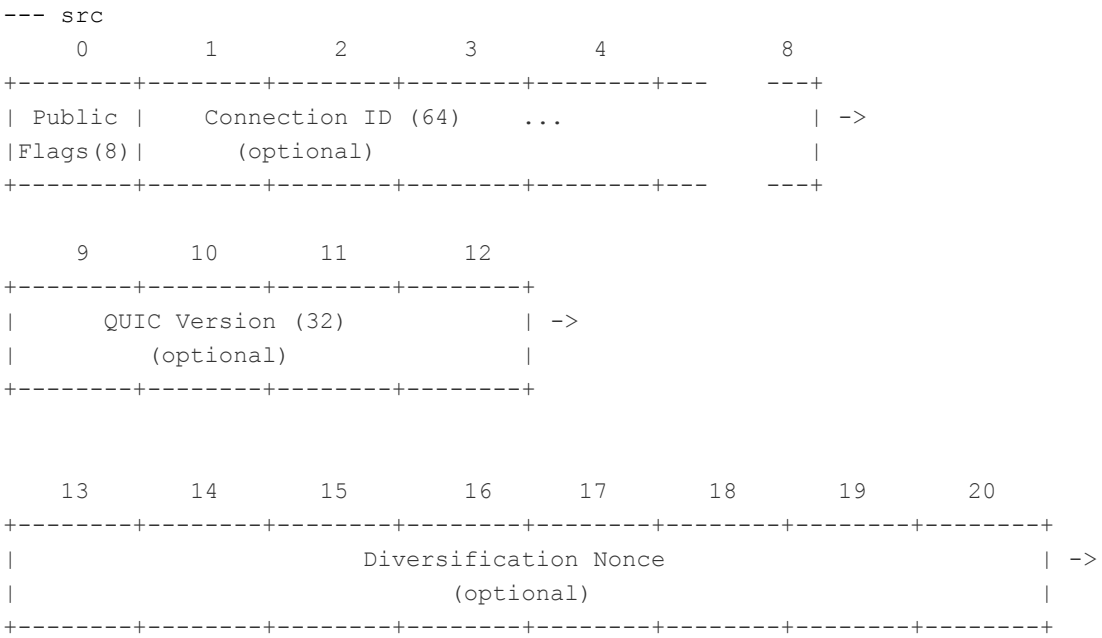
# Packet Types and Formats

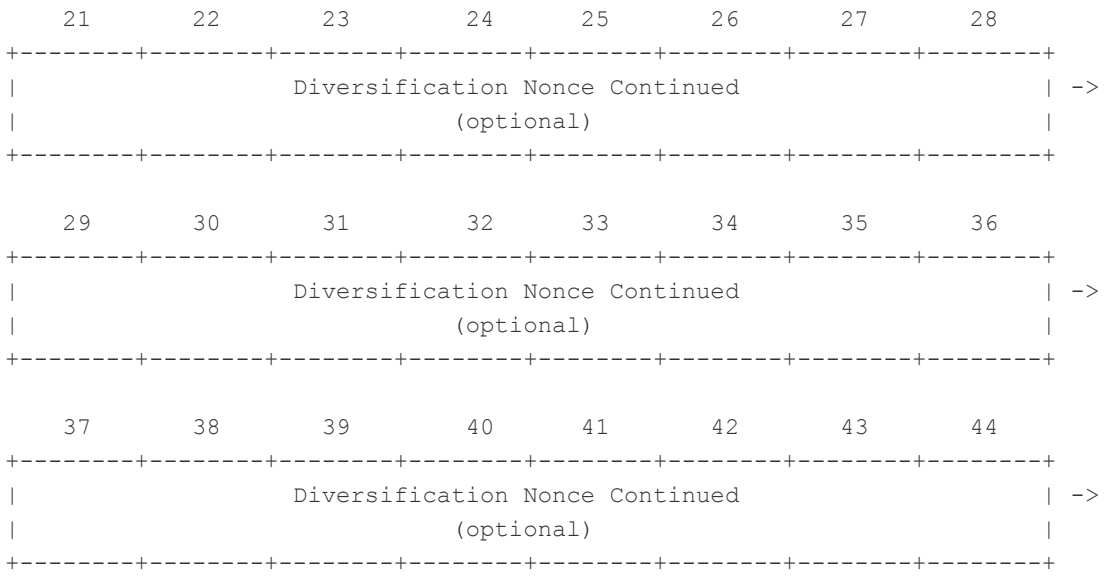
QUIC has Special Packets and Regular Packets. There are two types of Special Packets: Version Negotiation Packets and Public Reset Packets, and regular packets containing frames.

All QUIC packets should be sized to fit within the path's MTU to avoid IP fragmentation. Path MTU discovery is a work in progress, and the current QUIC implementation uses a 1350-byte maximum QUIC packet size for IPv6, 1370 for IPv4.  Both sizes are without IP and UDP overhead.

### QUIC Public Packet Header

All QUIC packets on the wire begin with a public header sized between 1 and 51 bytes. The wire format for the public header is as follows:






---

The payload may include various type-dependent header bytes as described below.
The fields in the public header are the following:

- **Public Flags:**
  - 0x01 = PUBLIC_FLAG_VERSION. Interpretation of this flag depends on whether the packet is sent by the server or the client. When sent by the client, setting it indicates that the header contains a QUIC Version (see below). This bit must be set by a client in all packets until confirmation from the server arrives agreeing to the proposed version is received by the client. A server indicates agreement on a version by sending packets without setting this bit. When this bit is set by the server, the packet is a Version Negotiation Packet. Version Negotiation is described in more detail later.
  - 0x02 = PUBLIC_FLAG_RESET. Set to indicate that the packet is a Public Reset packet.
  - 0x04 = Indicates the presence of a 32 byte diversification nonce in the header.
  - 0x08 = Indicates the full 8 byte Connection ID is present in the packet. This must be set in all packets until negotiated to a different value for a given direction (e.g., client may request fewer bytes of the Connection ID be presented).
  - Two bits at 0x30 indicate the number of low-order-bytes of the packet number that are present in each packet. The bits are only used for Frame Packets. For Public Reset and Version Negotiation Packets (sent by the server) which don't have a packet number, these bits are not used and must be set to 0. Within this 2 bit mask:
    - 0x30 indicates that 6 bytes of the packet number is present
    - 0x20 indicates that 4 bytes of the packet number is present
    - 0x10 indicates that 2 bytes of the packet number is present
    - 0x00 indicates that 1 byte of the packet number is present

    - 0x40 is reserved for multipath use.
    - 0x80 is currently unused, and must be set to 0.

- **Connection ID**: This is an unsigned 64 bit statistically random number selected by the client that is the identifier of the connection. Because QUIC connections are designed to remain established even if the client roams, the IP 4-tuple (source IP, source port, destination IP, destination port) may be insufficient to identify the connection. For each transmission direction, when the 4-tuple is sufficient to identify the connection, the connection ID may be omitted.

- **QUIC Version**: A 32 bit opaque tag that represents the version of the QUIC protocol. Only present if the public flags contain FLAG_VERSION (i.e public_flags & FLAG_VERSION !=0). A client may set this flag, and include EXACTLY one proposed version, as well as including arbitrary data (conforming to that version). A server may set this flag when the client-proposed version was unsupported, and may then provide a list (0 or more) of acceptable versions, but MUST not include any data after the version(s). Examples of version values in recent experimental versions include "Q025" which corresponds to byte 9 containing 'Q", byte 10 containing '0", etc. [See list of changes in various versions listed at the end of this document.]

- **Packet Number**: The lower 8, 16, 32, or 48 bits of the packet number, based on which FLAG_?BYTE_SEQUENCE_NUMBER flag is set in the public flags. Each Regular Packet (as opposed to the Special public reset and version negotiation packets) is assigned a packet number by the sender. The first packet sent by an endpoint shall have a packet number of 1, and each subsequent packet shall have a packet number one larger than that of the previous packet.

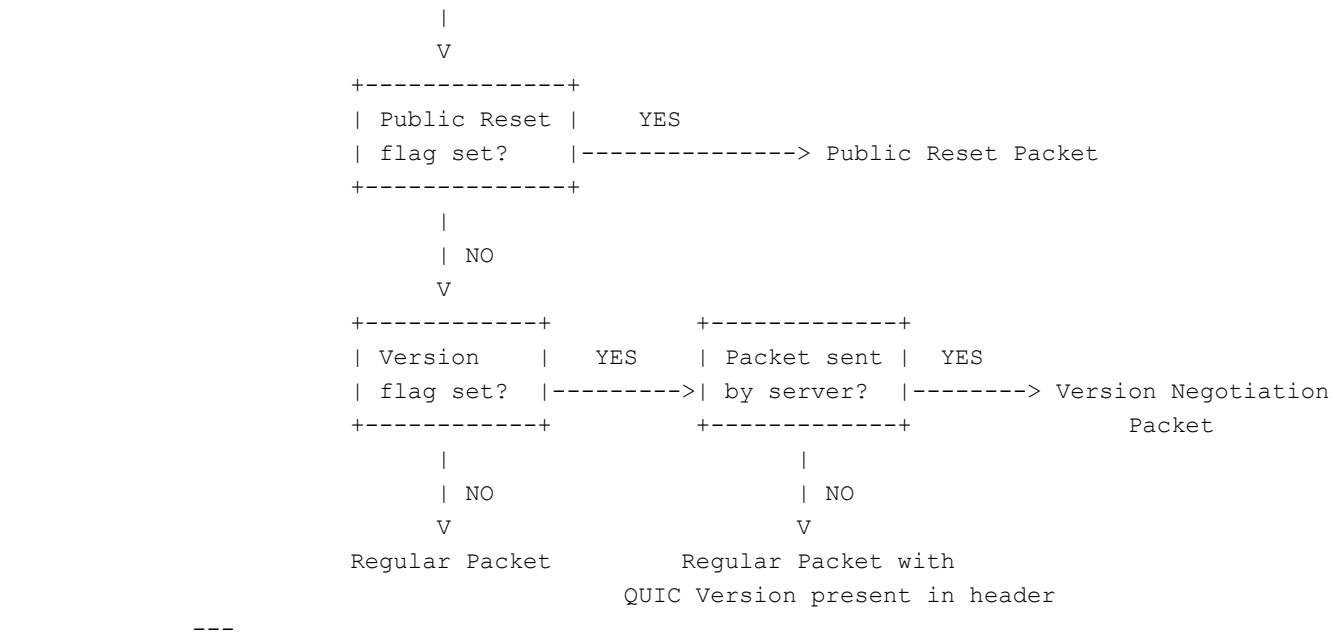
  The lower 64 bits of the packet number is used as part of a cryptographic nonce; therefore, a QUIC endpoint must not send a packet with a packet number that cannot be represented in 64 bits. If a QUIC endpoint transmits a packet with a packet number of (2^64-1), that packet must include a CONNECTION_CLOSE frame with an error code of QUIC_SEQUENCE_NUMBER_LIMIT_REACHED, and the endpoint must not transmit any additional packets.

  At most the lower 48 bits of a packet number are transmitted. To enable unambiguous reconstruction of the packet number by the receiver, a QUIC endpoint must not transmit a packet whose packet number is larger by (2^(bitlength-2)) than the largest packet number for which an acknowledgement is known to have been transmitted by the receiver. Therefore, there must never be more than (2^46) packets in flight.

  Any truncated packet number shall be inferred to have the value closest to the one more than the largest known packet number of the endpoint which transmitted the packet that originally contained the truncated packet number. The transmitted portion of the packet number matches the lowest bits of the inferred value.

A Public Flags processing flowchart follows:

```
--- src
Check the public flags in public header
                  |
```

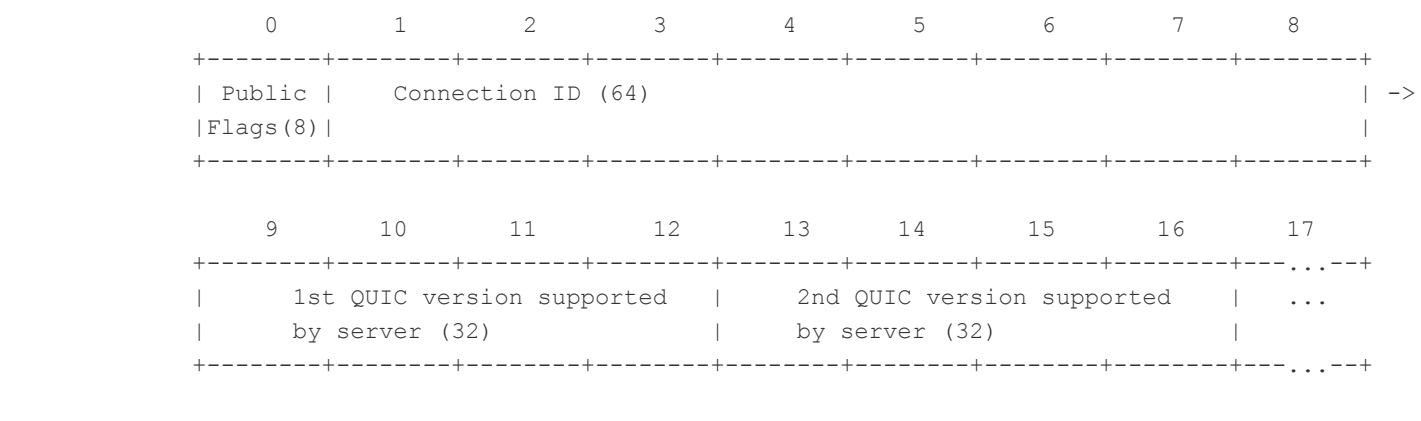



---

## Special Packets

### *Version Negotiation Packet*

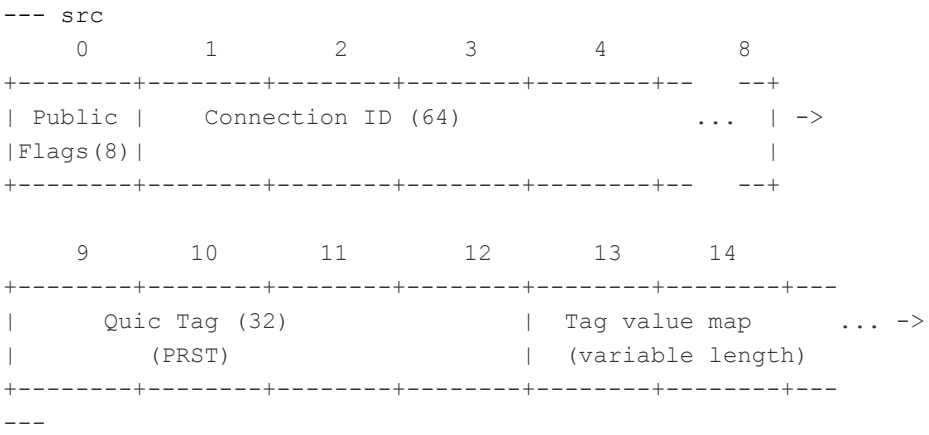
A version negotiation packet is only sent by the server. Version Negotiation packets begin with an 8-bit public flags and 64-bit Connection ID. The public flags must set PUBLIC_FLAG_VERSION and indicate the 64-bit Connection ID. The rest of the Version Negotiation packet is a list of 4-byte versions which the server supports:




---

### *Public Reset Packet*

A Public Reset packet begins with an 8-bit public flags and 64-bit Connection ID. The public flags must set PUBLIC_FLAG_RESET and indicate the 64-bit Connection ID. The rest of the Public Reset packet is encoded as if it were a crypto handshake message of the tag PRST (see [QUIC-CRYPTO]):




**Tag value map**: The tag value map contains the following tag-values:

- RNON (public reset nonce proof) - a 64-bit unsigned integer. Mandatory.
- RSEQ (rejected packet number) - a 64-bit packet number. Mandatory.
- CADR (client address) - the observed client IP address and port number. This is currently for debugging purposes only and hence is optional.

(TODO: Public Reset packet should include authenticated (destination) server IP/port.)

### Regular Packets

Regular Packets are authenticated and encrypted. The Public Header is authenticated but not encrypted, and the rest of the packet starting with the first frame is encrypted. Immediately following the Public Header, Regular Packets contain AEAD (authenticated encryption and associated data) data. This data must be decrypted in order for the contents to be interpreted. After decryption, the plaintext consists of a sequence of frames.

(TODO: Document the inputs to encryption and decryption and describe trial decryption.)

#### *Frame Packet*

Frame Packets have a payload that is a series of type-prefixed frames. The format of frame types is defined later in this document, but the general format of a Frame Packet is as follows:


# Life of a QUIC Connection

### Connection Establishment

A QUIC client is the endpoint that initiates a connection. QUIC's connection establishment intertwines

version negotiation with the crypto and transport handshakes to reduce connection establishment latency. We first describe version negotiation below.

Each of the initial packets sent from the client to the server must set the version flag, and must specify the version of the protocol being used. Every packet sent by the client must have the version flag on, until it receives a packet from the server with the version flag off. After the server receives the first packet from the client with the version flag off, it must ignore any (possibly delayed) packets with the version flag on.

When the server receives a packet with a Connection ID for a new connection, it will compare the client's version to the versions it supports. If the client's version is acceptable to the server, the server will use this protocol version for the lifetime of the connection. In this case, all packets sent by the server will have the version flag off.

If the client's version is not acceptable to the server, a 1-RTT delay will be incurred. The server will send a Version Negotiation Packet to the client. This packet will have the version flag set and will include the server's set of supported versions.

When the client receives a Version Negotiation Packet from the server, it will select an acceptable protocol version and resend all packets using this version. These packet must continue to have the version flag set and must include the new negotiated protocol version. Eventually, the client receives the first Regular Packet (i.e. not a Version Negotiation Packet) from the server indicating the end of version negotiation, and the client now sends all subsequent packets with the version flag off.

In order to avoid downgrade attacks, the version of the protocol that the client specified in the first packet and the set of versions supported by the server must be included in the crypto handshake data. The client needs to verify that the server's version list from the handshake matches the list of versions in the Version Negotiation Packet. The server needs to verify that the client's version from the handshake represents a version of the protocol that it does not actually support.

The rest of the connection establishment is described in the handshake document [QUIC-CRYPTO]. The crypto handshake is performed over the dedicated crypto stream (Stream ID 1).

During connection establishment, the handshake must negotiate various transport parameters. The currently defined transport parameters are described later in the document.

## Data Transfer

QUIC implements connection reliability, congestion control, and flow control. QUIC flow control closely follows HTTP/2's flow control. QUIC reliability and congestion control are described in an accompanying document. A QUIC connection uses a single packet sequence number space for shared congestion control and loss recovery across the connection.

All data transferred in a QUIC connection, including the crypto handshake, is sent as data inside streams, but the ACKs acknowledge QUIC Packets.

This section conceptually describes the use of streams for data transfer within a QUIC connection. The various frames that are mentioned in this section are described in the section on Frame Types and Formats.

### *Life of a QUIC Stream*

Streams are independent sequences of bi-directional data cut into stream frames. Streams can be created either by the client or the server, can concurrently send data interleaved with other streams, and can be cancelled. QUIC's stream lifetime is modeled closely after HTTP/2's [RFC7540]. (HTTP/2's usage of QUIC streams is described in more detail later in the document.)

Stream creation is done implicitly, by sending a STREAM frame for a given stream. To avoid stream ID collision, the Stream-ID must be even if the server initiates the stream, and odd if the client initiates the stream. 0 is not a valid Stream-ID. Stream 1 is reserved for the crypto handshake, which should be the first client-initiated stream. When using HTTP/2 over QUIC, Stream 3 is reserved for transmitting compressed headers for all other streams, ensuring reliable in-order delivery and processing of headers.

Stream-IDs from each side of the connection must increase monotonically as new streams are created. E.g. Stream 2 may be created after stream 3, but stream 7 must not be created after stream 9. The peer may receive streams out of order. For example, if a server receives packet 10 including frames for stream 9 before it receives packet 9 including frames for stream 7, it should handle this gracefully.

If the endpoint receiving a STREAM frame does not want to accept the stream, it can immediately respond with a RST_STREAM frame (described below). Note, however, that the initiating endpoint may have already sent data on the stream as well; this data must be ignored.

Once a stream is created, it can be used to send and receive data. This means that a series of stream frames can be sent by a QUIC endpoint on a stream until the stream is terminated in that direction.

Either QUIC endpoint can terminate a stream normally. There are three ways that streams can be terminated:

1. **Normal termination:** Since streams are bidirectional, streams can be "half-closed" or "closed". When one side of the stream sends a frame with the FIN bit set to true, the stream is considered to be "half-closed" in that direction. A FIN indicates that no further data will be sent from the sender of the FIN on this stream. When a QUIC endpoint has both sent and received a FIN, the endpoint considers the stream to be "closed". While the FIN should be sent with the last user data for a stream, the FIN bit can be sent on an empty stream frame following the last data on the stream.

2. **Abrupt termination:** Either the client or server can send a RST_STREAM frame for a stream at any time. A RST_STREAM frame contains an error code to indicate the reason for failure (error codes are listed later in the document.) When a RST_STREAM frame is sent from the stream originator, it indicates a failure to complete the stream and that no further data will be sent on the stream. When a RST_STREAM frame is sent from the stream receiver, the sender, upon receipt, should stop sending any data on the stream. The stream receiver should be aware that there is a race between data already in transit from the sender and the time the RST_STREAM frame is received. In order to ensure that the connection-level flow control is correctly accounted, even if a RST_STREAM frame is received, a sender needs to ensure that either: the FIN and all bytes in the stream are received by the peer or a RST_STREAM frame is received by the peer. This also means that the sender of a RST_STREAM frame needs to continue responding to incoming STREAM_FRAMEs on this stream with the appropriate

WINDOW_UPDATEs to ensure that the sender does not get flow control blocked attempting to deliver the FIN.

3. Streams are also terminated when the connection is terminated, as described in the next section.

### Connection Termination

Connections should remain open until they become idle for a pre-negotiated period of time. When a server decides to terminate an idle connection, it should not notify the client to avoid waking up the radio on mobile devices. A QUIC connection, once established, can be terminated in one of two ways:

1. **Explicit Shutdown:** An endpoint sends a CONNECTION_CLOSE frame to the peer initiating a connection termination. An endpoint may send a GOAWAY frame to the peer prior to a CONNECTION_CLOSE to indicate that the connection will soon be terminated. A GOAWAY frame when sent signals to the peer that any active streams will continue to be processed, but the sender of the GOAWAY will not initiate any additional streams and will not accept any new incoming streams. On termination of the active streams, a CONNECTION_CLOSE may be sent. If an endpoint sends a CONNECTION_CLOSE frame while unterminated streams are active (no FIN bit or RST_STREAM frames have been sent or received for one or more streams), then the peer must assume that the streams were incomplete and were abnormally terminated.

2. **Implicit Shutdown:** The default idle timeout for a QUIC connection is 30 seconds, and is a required parameter("ICSL") in connection negotiation. The maximum is 10 minutes. If there is no network activity for the duration of the idle timeout, the connection is closed. By default a CONNECTION_CLOSE frame will be sent. A silent close option can be enabled when it is expensive to send an explicit close, such as mobile networks that must wake up the radio.

An endpoint may also send a PUBLIC_RESET packet at any time during the connection to abruptly terminate an active connection. A PUBLIC_RESET is the QUIC equivalent of a TCP RST.

## Frame Types and Formats

QUIC Frame Packets are populated by frames. which have a Frame Type byte, which itself has a type-dependent interpretation, followed by type-dependent frame header fields. All frames are contained within single QUIC Packets and no frame can span across a QUIC Packet boundary.

### Frame Types

There are two interpretations for the Frame Type byte, resulting in two frame types: Special Frame Types, and Regular Frame Types. Special Frame Types encode both a Frame Type and corresponding flags all in the Frame Type byte, while Regular Frame Types use the Frame Type byte simply.

Currently defined Special Frame Types are:

```
--- src
   +------------------+-----------------------------+
   | Type-field value |     Control Frame-type      |
   +------------------+-----------------------------+
   |     1fdooossB    |  STREAM                     |
   |     01ntllmmB    |  ACK                        |
```

```
   |     001xxxxxB    |  CONGESTION_FEEDBACK        |
   +------------------+-----------------------------+
---
```

Currently defined Regular Frame Types are:

```
--- src
   +------------------+-----------------------------+
   | Type-field value |     Control Frame-type      |
   +------------------+-----------------------------+
   | 00000000B (0x00) |  PADDING                    |
   | 00000001B (0x01) |  RST_STREAM                 |
   | 00000010B (0x02) |  CONNECTION_CLOSE           |
   | 00000011B (0x03) |  GOAWAY                     |
   | 00000100B (0x04) |  WINDOW_UPDATE              |
   | 00000101B (0x05) |  BLOCKED                    |
   | 00000110B (0x06) |  STOP_WAITING               |
   | 00000111B (0x07) |  PING                       |
   +------------------+-----------------------------+
---
```

### STREAM Frame

The STREAM frame is used to both implicitly create a stream and to send data on it, and is as follows:

```
--- src
     0        1       …               SLEN
+--------+--------+--------+--------+--------+
|Type (8)| Stream ID (8, 16, 24, or 32 bits) |
|        |    (Variable length SLEN bytes)   |
+--------+--------+--------+--------+--------+

  SLEN+1  SLEN+2     …                                          SLEN+OLEN
+--------+--------+--------+--------+--------+--------+--------+--------+
|   Offset (0, 16, 24, 32, 40, 48, 56, or 64 bits) (variable length)    |
|                    (Variable length: OLEN  bytes)                     |
+--------+--------+--------+--------+--------+--------+--------+--------+

  SLEN+OLEN+1   SLEN+OLEN+2
+-------------+-------------+
| Data length (0 or 16 bits)|
|  Optional(maybe 0 bytes)  |
+------------+--------------+
---
```

The fields in the STREAM frame header are as follows:

- **Frame Type:** The Frame Type byte is an 8-bit value containing various flags (1fdooossB):
    - The leftmost bit must be set to 1 indicating that this is a STREAM frame.
    - The 'f' bit is the FIN bit. When set to 1, this bit indicates the sender is done sending on this stream and wishes to "half-close" (described in more detail later.)
    - which is described in more detail later in this document.
    - The 'd' bit indicates whether a Data Length is present in the STREAM header. When set

to 0, this field indicates that the STREAM frame extends to the end of the Packet.
    - The next three 'ooo' bits encode the length of the Offset header field as 0, 16, 24, 32, 40, 48, 56, or 64 bits long.
    - The next two 'ss' bits encode the length of the Stream ID header field as 8, 16, 24, or 32 bits long.
- **Stream ID:** A variable-sized unsigned ID unique to this stream.
- **Offset**: A variable-sized unsigned number specifying the byte offset in the stream for this block of data.
- **Data length:** An optional 16-bit unsigned number specifying the length of the data in this stream frame. The option to omit the length should only be used when the packet is a "full-sized" Packet, to avoid the risk of corruption via padding.

A stream frame must always have either non-zero data length or the FIN bit set.

## ACK Frame

The ACK frame is sent to inform the peer which packets have been received, as well as which packets are still considered missing by the receiver (the contents of missing packets may need to be resent). The ack frame contains between 1 and 256 ack blocks. Ack blocks are ranges of acknowledged packets, similar to TCP's SACK blocks, but QUIC has no equivalent of TCP's cumulative ack point, because packets are retransmitted with new sequence numbers.

To limit the ACK blocks to the ones that haven't yet been received by the peer, the peer periodically sends STOP_WAITING frames that signal the receiver to stop acking packets below a specified sequence number, raising the "least unacked" packet number at the receiver. A sender of an ACK frame thus reports only those ACK blocks between the received least unacked and the reported largest observed packet numbers. It is recommended for the sender to send the most recent largest acked packet it has received in an ack as the stop waiting frame's least unacked value.

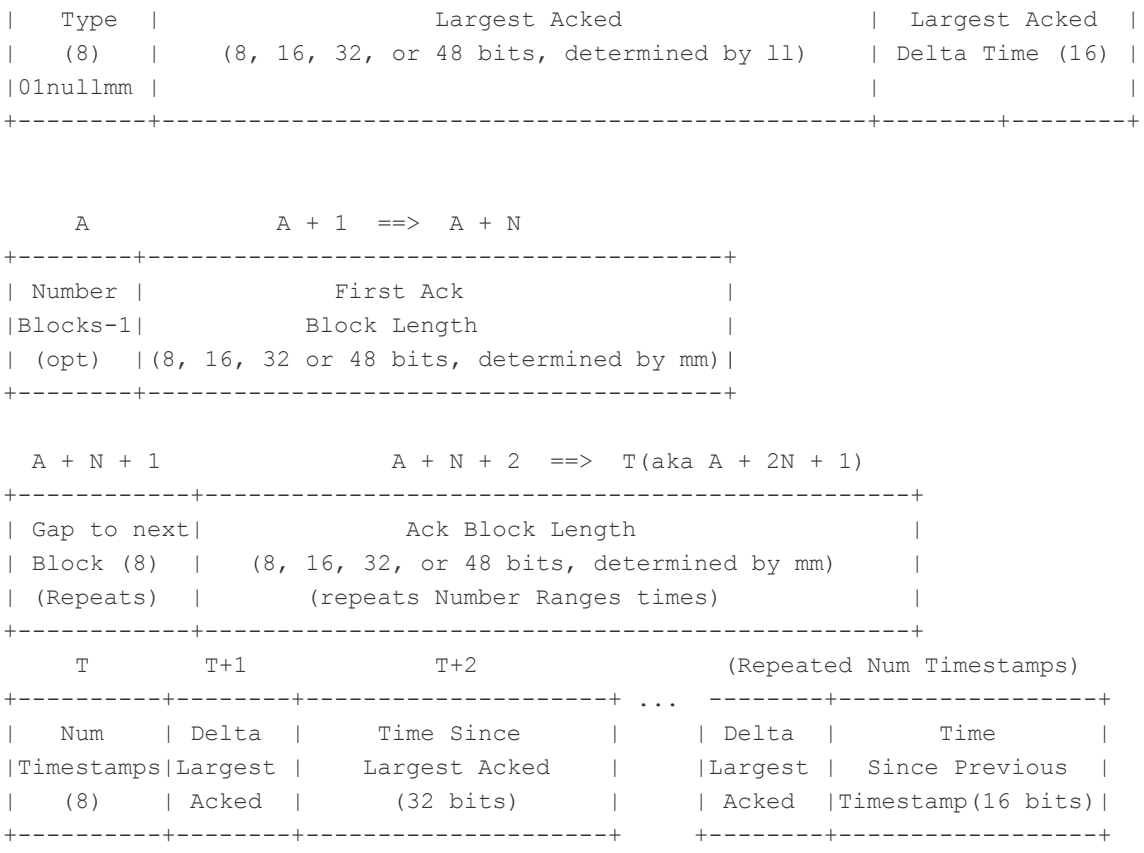
Unlike TCP SACKs, QUIC ACK blocks are irrevocable, so once a packet is acked, even if it does not appear in a future ack frame, it is assumed to be acked.

As a replacement for QUIC's deprecated entropy, the sender can intentionally skip packet numbers to introduce entropy into the connection. The sender must always close the connection if an unsent packet number is acked, so this mechanism automatically defeats any potential attackers. The ack format is efficient at expressing blocks of missing packets, so this has a low cost to the receiver and sender and efficiently provides up to 8 bits of entropy on demand, rather than incurring the constant overhead and achieving 8 bits of entropy. The 8 bits is the longest gap between ack ranges the ack format can efficiently express.

**Section Offsets**

**0**: Start of the ack frame.
**T**: Byte offset of the start of the timestamp section.
**A**: Byte offset of the start of the ack block section.
**N**: Length in bytes of the largest acked.

```
--- src
     0                            1  => N                       N+1 => A(aka N + 3)
+---------+--------------------------------------------------+--------+--------+
```




The fields in the ACK frame are as follows:

- **Frame Type:** The Frame Type byte is an 8-bit value containing various flags (01nullmmB).
    - The first two bits must be set to 01 indicating that this is an ACK frame.
    - The 'n' bit indicates whether the frame has more than 1 ack range.
    - The 'u' bit is unused.
    - The two 'll' bits encode the length of the Largest Observed field as 1, 2, 4, or 6 bytes long.
    - The two 'mm' bits encode the length of the Missing Packet Sequence Number Delta field as 1, 2, 4, or 6 bytes long.
- **Largest Acked**: A variable-sized unsigned value representing the largest packet number the peer has observed.
- **Largest Acked Delta Time**: A 16-bit unsigned float with 11 explicit bits of mantissa and 5 bits of explicit exponent, specifying the time elapsed in microseconds from when largest acked was received until this Ack frame was sent. The bit format is loosely modeled after IEEE 754. For example, 1 microsecond is represented as 0x1, which has an exponent of zero, presented in the 5 high order bits, and mantissa of 1, presented in the 11 low order bits. When the explicit exponent is greater than zero, an implicit high-order 12th bit of 1 is assumed in the mantissa. For example, a floating value of 0x800 has an explicit exponent of 1, as well as an explicit mantissa of 0, but then has an effective mantissa of 4096 (12th bit is assumed to be 1). Additionally, the actual exponent is one-less than the explicit exponent, and the value represents 4096 microseconds. Any values larger than the representable range are clamped to 0xFFFF.
- **Ack Block Section:**

    - **Num Blocks**: An optional 8-bit unsigned value specifying one less than the number of ack blocks. Only present if the 'n' flag bit is 1.
    - **Ack block length**: A variable-sized packet number delta. For the first missing packet range, the ack block starts at largest acked. For the first ack block, the length of the ack block is 1 + this value. For subsequent ack blocks, it is the length of the ack block. For non-first blocks, a value of 0 indicates more than 256 packets in a row were lost.
    - **Gap to next block**: An 8-bit unsigned value specifying the number of packets between ack blocks.
- **Timestamp Section:**
    - **Num Timestamp:** An 8-bit unsigned value specifying the number of timestamps that are included in this ack frame. There will be this many pairs of <packet number, timestamp> following in the timestamps.
    - **Delta Largest Observed:** An 8-bit unsigned value specifying the packet number delta from the first timestamp to the largest observed. Therefore, the packet number is the largest observed minus the delta largest observed.
    - **First Timestamp:** A 32-bit unsigned value specifying the time delta in microseconds, from the beginning of the connection of the arrival of the packet specified by Largest Observed minus Delta Largest Observed.
    - **Delta Largest Observed (Repeated):** (Same as above.)
    - **Time Since Previous Timestamp (Repeated):** A 16-bit unsigned value specifying delta from the previous timestamp. It is encoded in the same format as the Ack Delay Time.

### STOP_WAITING Frame

The STOP_WAITING frame is sent to inform the peer that it should not continue to wait for packets with packet numbers lower than a specified value. The packet number is encoded in 1, 2, 4 or 6 bytes, using the same coding length as is specified for the packet number for the enclosing packet's header (specified in the QUIC Frame Packet's Public Flags field.) The frame is as follows:

```
--- src
     0        1        2        3         4       5       6
+--------+--------+--------+--------+--------+-------+-------+
|Type (8)|   Least unacked delta (8, 16, 32, or 48 bits)     |
|        |                       (variable length)           |
+--------+--------+--------+--------+--------+--------+------+
---
```

The fields in the STOP_WAITING frame are as follows:

- **Frame Type:** The Frame Type byte is an 8-bit value that must be set to 0x06 indicating that this is a STOP_WAITING frame.
- **Least Unacked Delta:** A variable length packet number delta with the same length as the packet header's packet number. Subtract it from the header's packet number to determine the least unacked. The resulting least unacked is the smallest packet number of any packet for which the sender is still awaiting an ack. If the receiver is missing any packets smaller than this value, the receiver should consider those packets to be irrecoverably lost.

### WINDOW_UPDATE Frame

The WINDOW_UPDATE frame is used to inform the peer of an increase in an endpoint's flow control

receive window. The stream ID can be 0, indicating this WINDOW_UPDATE applies to the connection level flow control window, or > 0 indicating that the specified stream should increase its flow control window. The frame is as follows:

An absolute byte offset is specified, and the receiver of a WINDOW_UPDATE frame may only send up to that number of bytes on the specified stream. Violating flow control by sending further bytes will result in the receiving endpoint closing the connection.

On receipt of multiple WINDOW_UPDATE frames for a specific stream ID, it is only necessary to keep track of the maximum byte offset.

Both stream and session windows start with a default value of 16 KB, but this is typically increased during the handshake. To do this, an endpoint should negotiate the SFCW (Stream Flow Control Window) and CFCW (Connection/Session Flow Control Window) parameters in the handshake. The value associated with each tag should be the number of bytes for initial stream window and initial connection window respectively.

The frame is as follows:

```
--- src
    0         1                 4        5                 12
+--------+--------+-- ... --+-------+--------+-- ... --+-------+
|Type(8) |    Stream ID (32 bits)   |  Byte offset (64 bits)   |
+--------+--------+-- ... --+-------+--------+-- ... --+-------+
---
```

The fields in the WINDOW_UPDATE frame are as follows:

- **Frame Type:** The Frame Type byte is an 8-bit value that must be set to 0x04 indicating that this is a WINDOW_UPDATE frame.
- **Stream ID:** ID of the stream whose flow control windows is being updated, or 0 to specify the connection-level flow control window.
- **Byte offset:** A 64-bit unsigned integer indicating the absolute byte offset of data which can be sent on the given stream. In the case of connection level flow control, the cumulative number of bytes which can be sent on all currently open streams.

### BLOCKED Frame

The BLOCKED frame is used to indicate to the remote endpoint that this endpoint is ready to send data (and has data to send), but is currently flow control blocked. This is a purely informational frame, which is extremely useful for debugging purposes. A receiver of a BLOCKED frame should simply discard it (after possibly printing a helpful log message). The frame is as follows:

```
--- src
     0        1        2        3         4
+--------+--------+--------+--------+--------+
|Type(8) |          Stream ID (32 bits)      |
+--------+--------+--------+--------+--------+
---
```

The fields in the BLOCKED frame are as follows:

- **Frame Type:** The Frame Type byte is an 8-bit value that must be set to 0x05 indicating that

this is a BLOCKED frame.

- **Stream ID:** A 32-bit unsigned number indicating the stream which is flow control blocked. A non-zero Stream ID field specifies the stream that is flow control blocked. When zero, the Stream ID field indicates that the connection is flow control blocked at the connection level.

### CONGESTION_FEEDBACK Frame

The CONGESTION_FEEDBACK frame is an experimental frame currently not used. It is intended to provide extra congestion feedback information outside the scope of the standard ack frame. A CONGESTION_FEEDBACK frame must have the first three bits of the Frame Type set to 001. The last 5 bits of the Frame Type field are reserved for future use.

### PADDING Frame

The PADDING frame pads a packet with 0x00 bytes. When this frame is encountered, the rest of the packet is expected to be padding bytes. The frame contains 0x00 bytes and extends to the end of the QUIC packet. A PADDING frame only has a Frame Type field, and must have the 8-bit Frame Type field set to 0x00.

### RST_STREAM Frame

The RST_STREAM frame allows for abnormal termination of a stream. When sent by the creator of a stream, it indicates the creator wishes to cancel the stream. When sent by the receiver of a stream, it indicates an error or that the receiver did not want to accept the stream, so the stream should be closed. The frame is as follows:

```
--- src
     0        1            4      5              12     8             16
+-------+--------+-- ... ----+--------+-- ... ------+-------+-- ... ------+
|Type(8)| StreamID (32 bits) | Byte offset (64 bits)| Error code (32 bits)|
+-------+--------+-- ... ----+--------+-- ... ------+-------+-- ... ------+
---
```

The fields in a RST_STREAM frame are as follows:

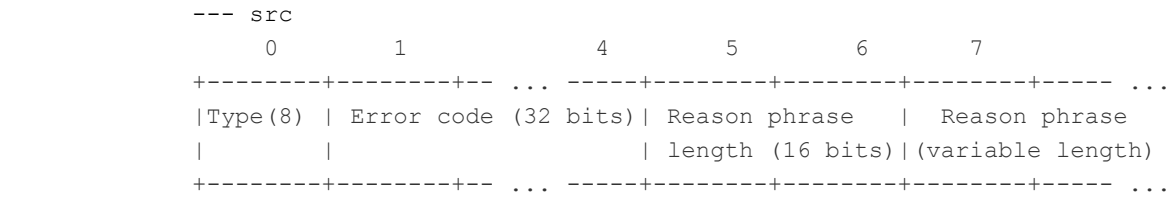
- **Frame type:** The Frame Type is an 8-bit value that must be set to 0x01 specifying that this is a RST_STREAM frame.
- **Stream ID:** The 32-bit Stream ID of the stream being terminated.
- **Byte offset:** A 64-bit unsigned integer indicating the absolute byte offset of the end of data for this stream.
- **Error code:** A 32-bit QuicErrorCode which indicates why the stream is being closed. QuicErrorCodes are listed later in this document.

### PING frame

The PING frame can be used by an endpoint to verify that a peer is still alive. The PING frame contains no payload. The receiver of a PING frame simply needs to ACK the packet containing this frame. The PING frame should be used to keep a connection alive when a stream is open. The default is to do this after 15 seconds of quiescence, which is much shorter than most NATs time out. A PING frame only has a Frame Type field, and must have the 8-bit Frame Type field set to 0x07.

### CONNECTION_CLOSE frame

The CONNECTION_CLOSE frame allows for notification that the connection is being closed. If there are streams in flight, those streams are all implicitly closed when the connection is closed. (Ideally, a GOAWAY frame would be sent with enough time that all streams are torn down.) The frame is as follows:

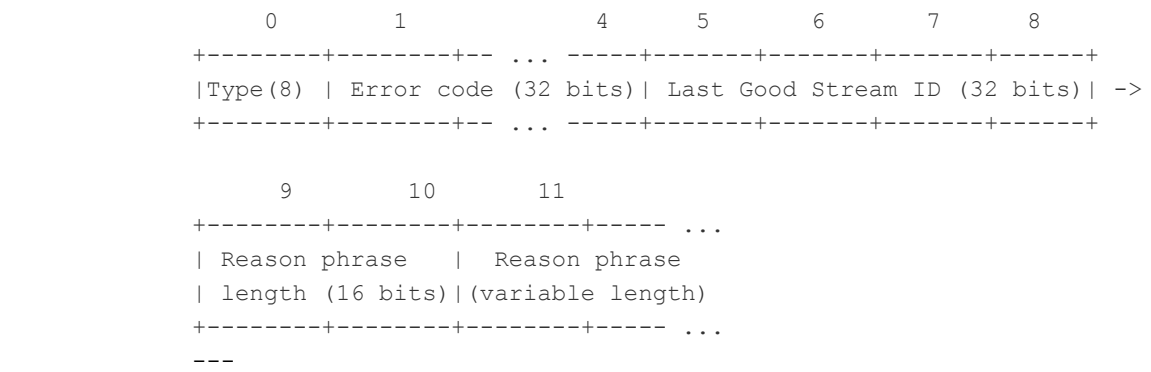



The fields of a CONNECTION_CLOSE frame are as follows:

- **Frame Type:** An 8-bit value that must be set to 0x02 specifying that this is a CONNECTION_CLOSE frame.
- **Error Code:** A 32-bit field containing the QuicErrorCode which indicates the reason for closing this connection.
- **Reason Phrase Length:** A 16-bit unsigned number specifying the length of the reason phrase. This may be zero if the sender chooses to not give details beyond the QuicErrorCode.
- **Reason Phrase:** An optional human-readable explanation for why the connection was closed.

### GOAWAY Frame

The GOAWAY frame allows for notification that the connection should stop being used, and will likely be aborted in the future. Any active streams will continue to be processed, but the sender of the GOAWAY will not initiate any additional streams, and will not accept any new streams. The frame is as follows:




The fields of a GOAWAY frame are as follows:

- **Frame type:** An 8-bit value that must be set to 0x03 specifying that this is a GOAWAY frame.
- **Error Code:** A 32-bit field containing the QuicErrorCode which indicates the reason for closing this connection.
- **Last Good Stream ID:** The last Stream ID which was accepted by the sender of the GOAWAY message. If no streams were replied to, this value must be set to 0.
- **Reason Phrase Length:** A 16-bit unsigned number specifying the length of the reason phrase. This may be zero if the sender chooses to not give details beyond the error code.
- **Reason Phrase:** An optional human-readable explanation for why the connection was closed.

# QUIC Transport Parameters

The handshake is responsible for negotiating a variety of transport parameters for a QUIC connection.

### Required Parameters

- **SFCW** - Stream Flow Control Window. The size in bytes of the stream level flow control window.

- **CFCW** - Connection Flow Control Window. The size in bytes of the connection level flow control window.

### Optional Parameters

- **SRBF** - Socket receive buffer size in bytes. The peer may want to limit their max CWND to something similar to the socket receive buffer if they fear the peer may sometimes be delayed in reading packets from kernel's socket buffer. Defaults to 256kbytes and has a minimum value of 16kbytes.

- **TCID** - Connection ID truncation. Indicates support for truncated Connection IDs. If sent by a peer, indicates the connection IDs sent to the peer should be truncated to 0 bytes. Useful for cases when a client ephemeral port is only used for a single connection.

- **COPT** - Connection Options are a repeated tag field. The field contains any connection options being requested by the client or server. These are typically used for experimentation and will evolve over time. Example use cases include changing congestion control algorithms and parameters such as initial window.

# QuicErrorCodes

The number to code mappings for QuicErrorCodes are currently defined in the Chromium source code in src/net/quic/quic_protocol.h. (TODO: hardcode numbers and add them here)

- QUIC_NO_ERROR: There was no error. This is not valid for RST_STREAM frames or CONNECTION_CLOSE frames
- QUIC_STREAM_DATA_AFTER_TERMINATION: There were data frames after the a fin or reset.
- QUIC_SERVER_ERROR_PROCESSING_STREAM: There was some server error which halted stream processing.
- QUIC_MULTIPLE_TERMINATION_OFFSETS: The sender received two mismatching fin or reset offsets for a single stream.
- QUIC_BAD_APPLICATION_PAYLOAD: The sender received bad application data.
- QUIC_INVALID_PACKET_HEADER: The sender received a malformed packet header.
- QUIC_INVALID_FRAME_DATA: The sender received an frame data. The more detailed error codes below are prefered where possible.
- QUIC_INVALID_FEC_DATA: FEC data is malformed.
- QUIC_INVALID_RST_STREAM_DATA: Stream rst data is malformed
- QUIC_INVALID_CONNECTION_CLOSE_DATA: Connection close data is malformed.
- QUIC_INVALID_ACK_DATA: Ack data is malformed.
- QUIC_DECRYPTION_FAILURE: There was an error decrypting.

- QUIC_ENCRYPTION_FAILURE: There was an error encrypting.
- QUIC_PACKET_TOO_LARGE: The packet exceeded MaxPacketSize.
- QUIC_PACKET_FOR_NONEXISTENT_STREAM: Data was sent for a stream which did not exist.
- QUIC_CLIENT_GOING_AWAY: The client is going away (browser close, etc.)
- QUIC_SERVER_GOING_AWAY: The server is going away (restart etc.)
- QUIC_INVALID_STREAM_ID: A stream ID was invalid.
- QUIC_TOO_MANY_OPEN_STREAMS: Too many streams already open.
- QUIC_CONNECTION_TIMED_OUT: We hit our pre-negotiated (or default) timeout
- QUIC_CRYPTO_TAGS_OUT_OF_ORDER: Handshake message contained out of order tags.
- QUIC_CRYPTO_TOO_MANY_ENTRIES: Handshake message contained too many entries.
- QUIC_CRYPTO_INVALID_VALUE_LENGTH: Handshake message contained an invalid value length.
- QUIC_CRYPTO_MESSAGE_AFTER_HANDSHAKE_COMPLETE: A crypto message was received after the handshake was complete.
- QUIC_INVALID_CRYPTO_MESSAGE_TYPE: A crypto message was received with an illegal message tag.
- QUIC_SEQUENCE_NUMBER_LIMIT_REACHED: Transmitting an additional packet would cause a packet number to be reused.

# Priority

(TODO: implement)
QUIC will use the HTTP/2 prioritization mechanism. Roughly, a stream may be dependent on another stream. In this situation, the "parent" stream should effectively starve the "child" stream. In addition, parent streams have an explicit priority. Parent streams should not starve other parent streams, but should make progress proportional to their relative priority.

# HTTP/2 Layering over QUIC

Since QUIC integrates various HTTP/2 mechanisms with transport mechanisms, QUIC implements a number of features that are also specified in HTTP/2. As a result, QUIC allows HTTP/2 mechanisms to be replaced by QUIC's implementation, reducing complexity in the HTTP/2 protocol. This section briefly describes how HTTP/2 semantics can be offered over a QUIC implementation.

### Stream Management

When HTTP/2 headers and data are sent over QUIC, the QUIC layer handles most of the stream management. HTTP/2 Stream IDs are replaced by QUIC Stream IDs. HTTP/2 does not need to do any explicit stream framing when using QUIC---data sent over a QUIC stream simply consists of HTTP/2 headers or body. Requests and responses are considered complete when the QUIC stream is closed in the corresponding direction.

Stream flow control is handled by QUIC, and does not need to be re-implemented in HTTP/2. QUIC's flow controller replaces the two levels of poorly matched flow controllers in current HTTP/2 deployments---one at the HTTP/2 level, and the other at the TCP level.

### HTTP/2 Header Compression

QUIC implements HPACK header compression for HTTP/2 [RFC7541], which unfortunately introduces some Head-of-Line blocking since HTTP/2 header blocks must be decompressed in the order they were compressed.

Since streams may be processed in arbitrary order at a receiver, strict ordering across headers is enforced by sending all headers on a dedicated headers stream, with Stream ID 3. An HTTP/2 receiver using QUIC would thus process data from a stream only after receiving the corresponding header on the headers stream.

Future work will tweak the compressor and decompressor in QUIC so that the compressed output does not depend on unacked previous compressed state. This could be done, perhaps, by creating "checkpoints" of HPACK state which are updated when headers have been acked. When compressing headers QUIC would only compress relative to the previous "checkpoint".

### Parsing HTTP/2 Headers

Bytes sent on the dedicated headers stream are simply HTTP/2 HEADERS frames. The exact layout of these frames is described in Section 6.2 of [RFC7540].

### QUIC Negotiation in HTTP

The Alternate-Protocol header is used to negotiate use of QUIC on future HTTP requests. To specify QUIC as an alternate protocol available on port 123, a server uses:

```
--- src
"Alternate-Protocol: 123:quic"
---
```

When a client receives a Alternate-Protocol header advertising QUIC, it can then attempt to use QUIC for future secure connections on that domain. Since middleboxes and/or firewalls can block QUIC and/or UDP communication, a client should implement a graceful fallback to TCP when QUIC reachability is broken.

Note that the server may reply with multiple field values or a comma-separated field value for Alternate-Protocol to indicate the various transports it supports.

A server can also send a header to notify that QUIC should not be used on this domain. If it sends the alternate-protocol-required header, the client should remember to not use QUIC on that domain in future, and not do any UDP probing to see if QUIC is available.

## Handshake Protocol Requirements

QUIC provides a dedicated stream (Stream ID 1) to be used for performing a combined connection and security handshake, but the details of this handshake protocol are out of this document's scope. However, QUIC does impose a number of requirements on any such handshake protocol. The following

list of requirements documents properties of the current prototype handshake which should be provided by any future handshake protocol.

### Connection Establishment in 0-RTT

The QUIC handshake protocol manages to successfully achieve 0-RTT for most connections, and is critical to QUIC's latency improvements.

### Source Address Spoofing Defense

TCP verifies the client's address by burning a round trip on the SYN, SYN_ACK exchange. QUIC uses a source address token delivered by the server in a previous connection.

### Opaque Source Address Tokens

QUIC servers store a number of pieces of data in the source address token, for use on a subsequent connection from the same client. This includes recently used source addresses, measured bandwidth to the client, and server-designated connection IDs (for Stateless REJs). An alternative handshake protocol's analog of a source address token needs to be (i) opaque at the client, and (ii) large enough to permit these bits of information to be stored. Alternatively, the handshake protocol should have a different method to store this information at the client.

### Transport Parameter Negotiation

In addition to negotiating crypto parameters, the QUIC handshake also negotiates QUIC and HTTP/2 level parameters, including max open QUIC streams and other QUIC connection options.

### Certificate Compression

The QUIC handshake compresses certificates so that an REJ, including the common Google certificate chain, is able to fit into two 1350 byte packets. This helps to reduce the amplification attack footprint of QUIC without reducing 0-RTT rate.

### Server Config Update

QUIC uses a Server Config Update (SCUP) message to refresh the source-address token (STK) and server config mid-connection, extending the period over which 0-RTT connections can be established by the client.

## Recent Changes By Version

- Q009: added priority as the first 4 bytes on spdy streams.
- Q010: renumber the various frame types
- Q011: shrunk the fnv128 hash on NULL encrypted packets from 16 bytes to 12 bytes.
- Q012: optimize the ack frame format to reduce the size and better handle ranges of nacks, which should make truncated acks virtually impossible. Also adding an explicit flag for truncated acks and moving the ack outside of the connection close frame.
- Q013: Compressed headers for *all* data streams are serialized into a reserved stream. This ensures serialized handling of headers, independent of stream cancellation notification.
- Q014: Added WINDOW_UPDATE and BLOCKED frames, no behavioral change.
- Q015: Removes the accumulated_number_of_lost_packets field from the TCP and inter arrival

congestion feedback frames and adds an explicit list of recovered packets to the ack frame.
- Q016: Breaks out the sent_info field from the ACK frame into a new STOP_WAITING frame.
- Changed GUID to Connection ID
- Q017: Adds stream level flow control
- Q018: Added a PING frame
- Q019: Adds session/connection level flow control
- Q020: Allow endpoints to set different stream/session flow control windows
- Q021: Crypto and headers streams are flow controlled (at stream level)
- Q023: Ack frames include packet timestamps
- Q024: HTTP/2-style header compression
- Q025: HTTP/2-style header keys. Removal of error_details from the RST_STREAM frame.
- Q026: Token binding, adds expected leaf cert (XLCT) tag to client hello
- Q027: Adds a nonce to the server hello
- Q029: Server and client honor QUIC_STREAM_NO_ERROR on early response
- Q030: Add server side support of certificate transparency.
- Q031: Adds a SHA256 hash of the serialized client hello messages to crypto proof.
- Q032: FEC related fields are removed from wire format.
- Q033: Adds an optional diversification nonce to packet headers, and eliminates the 2 byte and 4 byte connection ID length public flags.
- Q034: Removes entropy and private flags and changes the ack frame from nack ranges to ack ranges and removes truncated acks.
- Q035: Allows each endpoint to independently set maximum number of supported incoming streams using the MIDS ("Maximum Incoming Dynamic Streams") tag instead of the older MSPC ("Maximum Streams Per Connection") tag.
- Q036: Adds support for inducing head-of-line blocking between streams via the new FHOL tag in the handshake.

# Contributors

This protocol is the outcome of work by many engineers, not just the authors of this document. The design and rationale behind QUIC draw significantly from work by Jim Roskind. In alphabetical order, the contributors to the project are: Britt Cyr, Jeremy Dorfman, Ryan Hamilton, Jana Iyengar, Fedor Kouranov, Charles Krasic, Jo Kulik, Adam Langley, Jim Roskind, Robbie Shade, Satyam Shekhar, Cherie Shi, Ian Swett, Raman Tenneti, Victor Vasiliev, Antonio Vicente, Patrik Westin, Alyssa Wilk, Dale Worley, Fan Yang, Dan Zhang, Daniel Ziegler.

# Acknowledgments

Special thanks are due to the following for helping shape QUIC and its deployment: Chris Bentzel, Misha Efimov, Roberto Peon, Alistair Riddoch, Siddharth Vijayakrishnan, and Assar Westerlund. QUIC has also benefited immensely from discussions with folks in private conversations and public ones on the proto-quic@chromium.org mailing list.

.