QUIC Wire Layout Specification

This document is intended to mirror the rapidly evolving QUIC implementation found in the Chromium.org code repository. Due to the rapid development speed, this document may periodically be outdated. This is a very living document, as the QUIC protocol is also evolving based on results of experiments.

You may find the QUIC FAQ useful for a very brief overview. Alternatively, the rationalization, and justification for most of this wire layout specification can be found in the QUIC Design Document and Rationalization.

Detailed explanation of the cryptographic elements of QUIC are found in the QUIC Crypto document, which is generally very precise and up-to-date, but is continuing to undergo careful scrutiny and review.

TBD(jar) write a formal specification of QUIC.

Overview

QUIC aims to address a number of transport-layer and application-layer problems associated with modern web applications, while requiring little or no change from the perspective of web application writers. They include:

- **0-RTT**: In the common case, an encrypted request will be able to be sent from the client to the server without waiting for any communication from the server. SSL over TCP requires 2 or more full round trips before encrypted application data may be sent.
- **Recover Lost Packets**: QUIC can send periodic FEC packets which contain parity data that can allow lost packets to be recovered without needing to be retransmitted by the client. TCP requires the client to wait for a retry timeout to expire.
- **No Head of Line Blocking**: When a packet is lost (and not recovered by FEC), QUIC will continue to process subsequent packets. Since QUIC is a multiplexed protocol, this means that the only stream affected by packet loss is the stream whose data was lost. Other streams remain unaffected.
- **Better Congestion Control**: The flow control algorithms planned to be used by QUIC will pace the sending of packets to avoid quick bursts. Pacing has been shown to dramatically reduce the rate of packet loss when it matches the available bandwidth. Initial algorithms will be modeled after TCP's congestion control, such as TCP Cubic.

Definitions

- *Client*: The endpoint initiating the QUIC session.
- *Connection*: A bi-directional exchange of packets between a client and a server, identified by a GUID.
- *Endpoint*: Either the client or server of a connection.

- *GUID*: A unique identifier for a particular connection.
- *Packet*: A UDP packet containing framed data sent over a QUIC session.
- *Server*: The endpoint not initiating the QUIC session.
- *Session*: A sequence of packets sent over a single connection. This is the same as the "framing layer".
- *Stream*: A bi-directional flow of bytes across a virtual channel within a QUIC session.

Framing Layer

Connections

The QUIC framing layer (or "session") runs atop UDP, unlike how HTTP works today. The client is expected to be the QUIC session initiator. Because it runs on UDP, the underlying transport is not reliable, and the QUIC layer will handle retransmission of dropped packets for reliable streams. Unlike HTTP, the underlying connection for QUIC is guaranteed to be persistent. The HTTP “Connection” header therefore does not apply. For best performance, it is expected that clients will not close open sessions until the user navigates away from all web pages referencing a session, or until the server closes the sessions. Sessions should remain open until they become idle for a pre-negotiated period of time. When a server decides to terminate an idle session, it should not notify the client to avoid waking up the radio on mobile devices.

Framing

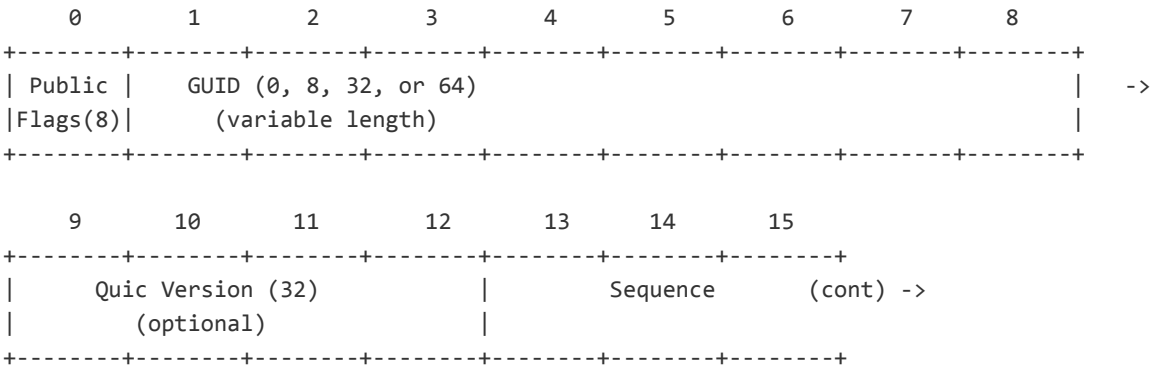
Once the session is established, clients and servers exchange UDP packets. There are two types of packets: data packets and FEC packets. Packets always have a common header which is between 3 and 21 bytes. FEC packets contain parity data which allows dropped packets to be recovered. Data packets contain stream payload data, ACK data, and control data (e.g. streams or session termination).

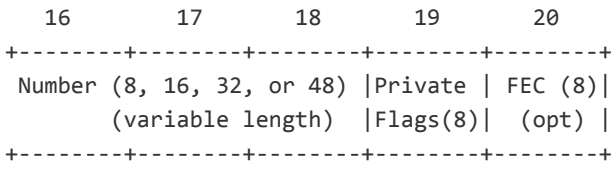
The simple header is designed to make reading and writing of packets easy.

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

QUIC Packet Header

All packets begin with a common header





Public Flags: An 8 bit value specifying per-packet public flags. Valid flags are:

- 0x00 = FLAG_DATA - signifies that this packet represents a data packet
- 0x01 = FLAG_VERSION - signifies that this packet also contains the version of the QUIC protocol.
- 0x02 = FLAG_RST - signifies that this packet is a public reset packet.
- 0x00 = FLAG_0BYTE_GUID - signifies the packet has a 0 byte truncated guid.
- 0x04 = FLAG_1BYTE_GUID - signifies the packet has a 1 byte truncated guid.
- 0x08 = FLAG_4BYTE_GUID - signifies the packet has a 4 byte truncated guid.
- 0x0C = FLAG_8BYTE_GUID - signifies the packet has a full 8 byte guid.
- 0x00 = FLAG_1BYTE_SEQUENCE_NUMBER - signifies the packet has a 1 byte truncated sequence number.
- 0x10 = FLAG_2BYTE_SEQUENCE_NUMBER - signifies the packet has a 2 byte truncated sequence number.
- 0x20 = FLAG_4BYTE_SEQUENCE_NUMBER - signifies the packet has a 4 byte truncated sequence number.
- 0x30 = FLAG_6BYTE_SEQUENCE_NUMBER - signifies the packet has a full 6 byte sequence number.

GUID (Globally Unique Identifier): This is an unsigned 64 bit pseudo random number selected by the client that is the key to the connection. Because QUIC sessions 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. When less uniqueness is sufficient to identify the connection, a truncated transmitted guid length is negotiable.

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).

Sequence Number: The lower 8, 16, 32, or 48 bits of the sequence number, based on which FLAG_?BYTE_SEQUENCE_NUMBER flag is set in the public flags. See “Sequence numbers” below.

Private Flags:

- 0x01 = FLAG_ENTROPY - for data packets, signifies that this packet contains the 1 bit of entropy, for fec packets, contains the xor of the entropy of protected packets.
- 0x02 = FLAG_FEC_GROUP - indicates whether the fec byte is present.
- 0x04 = FLAG_FEC - signifies that this packet represents an FEC packet.

FEC (FEC Group Number Offset): An FEC Group Number is the Packet Sequence Number of the first packet in the FEC group. The FEC Group Number Offset is an 8 bit unsigned value which should be subtracted from the current packet’s Packet Sequence Number to yield the FEC Group Number for this packet. This is only present if the private flags contain FLAG_FEC_GROUP.

Sequence numbers

Each QUIC packet is assigned a sequence number by the sender. The first packet sent by an endpoint shall have a sequence number of 1, and each subsequent packet shall have a sequence number one larger than that of the previous packet.

The lower 64 bits of the sequence number may be used as part of a cryptographic nonce; therefore, a QUIC endpoint must not send a packet with a sequence number that cannot be represented in 64 bits. If a QUIC endpoint transmits a packet with a sequence 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 sequence number are transmitted. To enable unambiguous reconstruction of the sequence number by the receiver, a QUIC endpoint must not transmit a packet whose sequence number is larger by (2^(bitlength-2)) than the largest sequence 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 sequence number shall be inferred to have the value closest to the one more than the largest known sequence number of the endpoint which transmitted the packet that originally contained the truncated sequence number. The transmitted portion of the sequence number matches the lowest bits of the inferred value.

Data Packets

Data packets (those packets with FLAG_DATA set) have a payload that is a series of type-prefixed frames. The format of frame types is defined later.

```
+--------+.........+--------+.........+
| Type   | Payload | Type   | Payload |
+--------+.........+--------+.........+
```

FEC Packets

FEC packets (those packets with FLAG_FEC set) have a payload that simply contains an XOR of the null-padded payload of each Data Packet in the FEC group.

```
+............+
| Redundancy |
+............+
```

Streams

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. The usage of streams with SPDY is that SPDY streams map to QUIC streams, using the QUIC stream ID, cancel using QUIC's RST_FRAME etc. See SPDY layering section for details.

Stream creation

Stream creation is done implicitly, by sending a STREAM_FRAME for a given stream. If the server is initiating the stream, the Stream-ID must be even. If the client is initiating the stream, the Stream-ID must be odd. 0 is not a valid Stream-ID. Stream 1 is reserved for the crypto handshake, which should be the first client-initiated stream.

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_FRAME. Note, however, that the initiating endpoint may have already sent data on the stream as well; this data must be ignored.

Stream data exchange

Once a stream is created, it can be used to send arbitrary amounts of data. Generally this means that a series of frames will be sent on the stream until a frame containing the fin bit is set. Once the FIN has been sent, the stream is considered to be half-closed. (See Stream half-close)

Stream half-close

When one side of the stream sends a frame with FIN set to true, the stream is considered to be half-closed from that side. The sender of the FIN is indicating that no further data will be sent from the sender on this stream. When both sides have half-closed, the stream is considered to be closed. (See Stream close)

Stream close

There are four ways that streams can be terminated:

1. **Normal termination**
   Normal stream termination occurs when both sender and receiver have half-closed the stream by sending a FIN. Ideally the FIN will be sent with the last data for the stream, for optimal byte packing.
2. **Abrupt termination**
   Either the client or server can send a RST_FRAME for a stream at any time. A RST_FRAME contains an error code to indicate the reason for failure. When a RST_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_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_FRAME is received.
3. **QUIC connection teardown**
   If the QUIC connection is torn down via a CONNECTION_CLOSE_FRAME while unterminated streams are active (no RST_STREAM frames have been sent or received for the stream), then the endpoint must assume that the stream was abnormally interrupted and may be incomplete.
4. **Implicit timeout**
   The default idle timeout for a QUIC connection is 10 minutes, unless a different timeout is negotiated at the start of the connection. If there is no network activity for the duration of the idle

timeout, the connection is implicitly closed. No CONNECTION_CLOSE_FRAME will be sent in this case, as it's expensive for mobile networks to wake the radio up in order to notify that the connection is done.

Frames

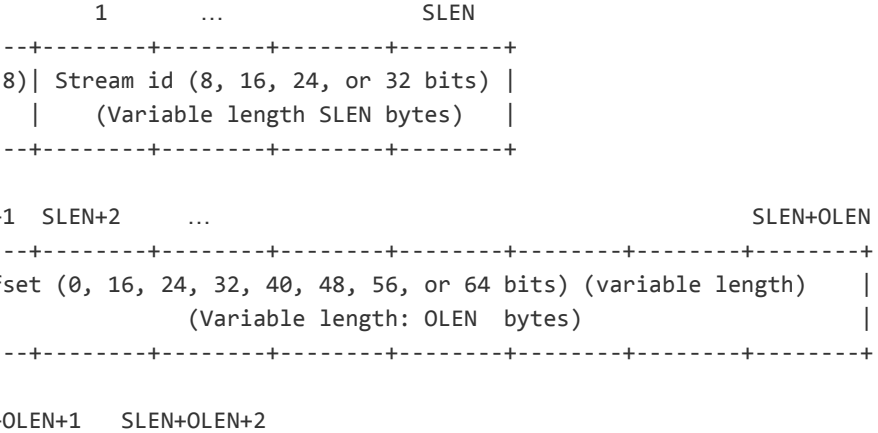
Frame types

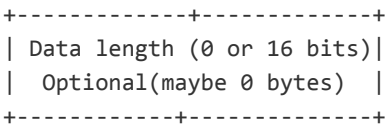
Type-field value	Control Frame-type
fdooooss0B	STREAM_FRAME
00000001B	ACK_FRAME
00000011B	CONGESTION_FEEDBACK_FRAME
00000111B	PADDING_FRAME
00100111B	RST_STREAM_FRAME
00101111B	CONNECTION_CLOSE_FRAME
00101011B	GOAWAY_FRAME

STREAM_FRAME

The STREAM_FRAME is used to both implicitly create a stream and to send data on it.

The type-field value of the stream frame contains flags to encode the StreamID, Offset, Data Length, and Fin, read from right to left.



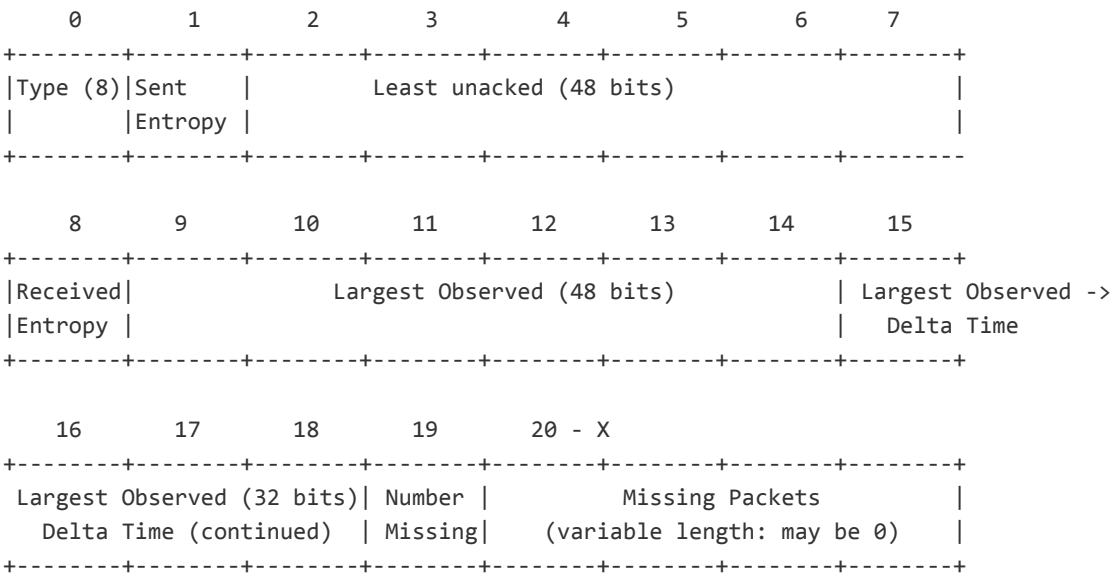


Frame type: an 8 bit value specifying that this is a stream frame as well as specifying the length of the stream id, the offset length, whether there is a data length, and whether it's a fin, from right to left.
Stream Id: An 8, 16, 24, or 32 bit unsigned ID unique to this stream. The encoded length is specified by the two bits in the type-field value, with 00 corresponding to 8 bits and 11 corresponding to 32 bits.
Fin: The fin bit is set to 0 if the sender may send more data on this stream. 1 indicates the sender is done sending on this stream and wishes to half-close.
Offset: A 0, 16, 24, 32, 40, 48, 56, or 64 bit unsigned number specifying the byte offset in the stream for this block of data. The encoded length is specified by the three offset bits in the type-field value, with 000 corresponding to 0 bits and 111 corresponding to 64 bits.
Data length: An optional uint16 representing length of the data following this stream frame. This length is present when the type byte is 1, and otherwise the rest of the packet contains data for the stream.

A stream frame MUST have non-zero data length, or have 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).



Frame type: Each ack frame starts with an 8 bit value specifying that this is a ack frame (?????001B). The first 5 bits are reserved for future use.

Data in an ACK frame is divided into two sections:

Sent Packet Data

Sent Entropy: An 8 bit unsigned value specifying the cumulative hash of entropy in all sent packets up to the packet with sequence number one less than the least unacked packet.
Least Unacked: The lower 48 bits of the smallest sequence 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.

Received Packet Data

Received Entropy: An 8 bit unsigned value specifying the cumulative hash of entropy in all received packets up to the largest observed packet.
Largest Observed: If the value of Missing Packets includes every packet observed to be missing since the last ACK_FRAME transmitted by the sender, then this value shall be the lower 48 bits of the largest observed sequence number. If there are packets known to be missing which are not present in Missing Packets (due to size limitations), then this value shall be the lower 48 bits of the largest sequence number smaller than the first missing packet which this ack does not include. (If multiple consecutive packets are lost, the value of Largest Observed may also appear in Missing Packets.)
Largest Observed Delta Time: A 32 bit unsigned value specifying the time elapsed in microseconds from when largest observed was received until this Ack frame was sent.
Num Missing: An 8 bit unsigned value specifying the number of missing packets between largest observed and least unacked.
Missing Packets: A series of the lower 48 bits of the sequence numbers of packets which have not yet been received.

CONGESTION_FEEDBACK_FRAME

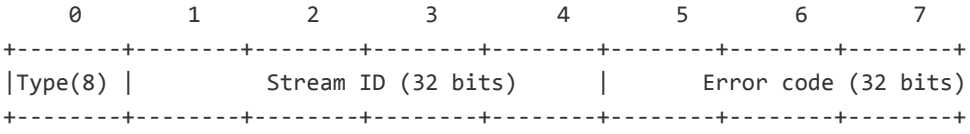
The CONGESTION_FEEDBACK_FRAME is is sent to inform the peer which packets have been received, as well as which packets will be resent.

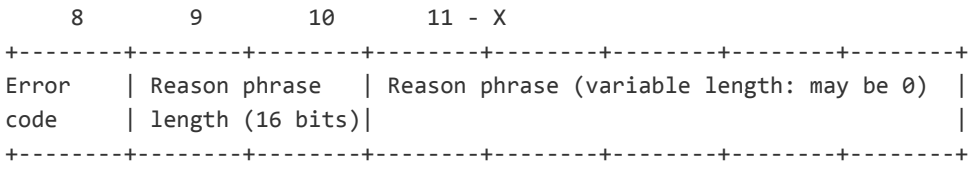
Frame type: Each congestion control frame starts with an 8 bit value specifying that this is a congestion control frame (?????011B). The first 5 bits are reserved for future use.

Congestion Control Data

Congestion Control Feedback: A sequence of bytes whose length and value depends on the particular choice of congestion control algorithm. See “Pacing and Congestion Control” below.

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.

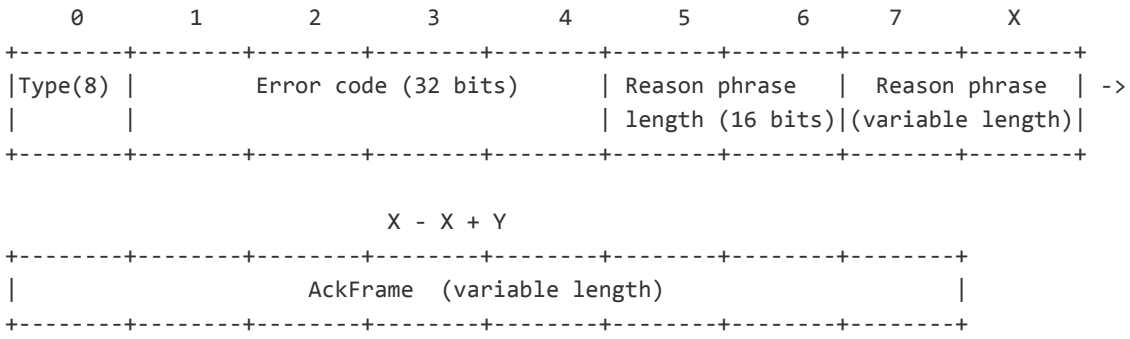




Frame type: an 8 bit value specifying that this is a stream rst frame (0x4)
Stream Id: A 32 bit ID unique to this stream.
Error code: A 32-bit QuicErrorCode which indicates why the stream is being closed.
Reason phrase length: A 16-bit unsigned integer specifying the length of the reason phrase. This may be zero if the sender chooses to not give details beyond the error code.
Reason phrase: A UTF-8 encoded optional human-readable explanation for why the connection was closed.

CONNECTION_CLOSE_FRAME

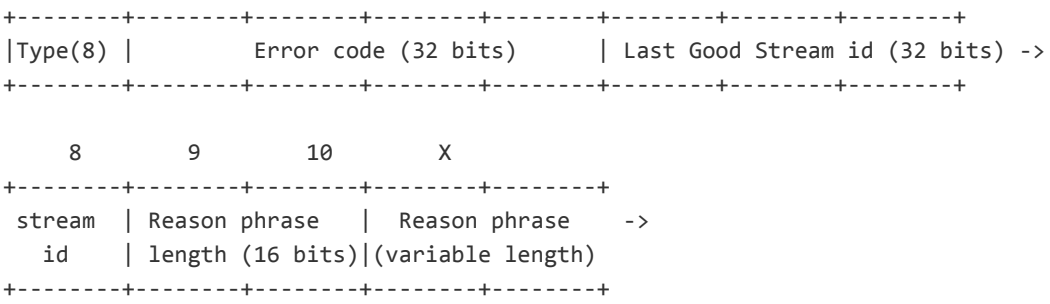
The CONNECTION_CLOSE_FRAME allows for notification that the connection is being aborted: it can be compared to closing a TCP connection. If there are streams in flight, those streams are all implicitly closed when the connection is closed. Ideally, a GOAWAY_FRAME would be sent with enough time that all streams are torn down.



Frame type: an 8 bit value specifying that this is a connection close frame (0x5)
Error code: 32 bits containing the QuicErrorCode which indicates why the connection is being closed.
Reason phrase length: a uint16 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.
AckFrame: A final ack frame, letting the peer know which packets had been received at the time the connection was closed.

GOAWAY_STREAM_FRAME

The GOAWAY_CLOSE_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.



Frame type: an 8 bit value specifying that this is a goaway frame (0x6)
Error code: 32 bits containing the QuicErrorCode which indicates why the connection is being closed.
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 0.
Reason phrase: An optional human-readable explanation for why the connection was closed.

Reason phrase length: a uint16 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.

QuicErrorCodes

The number to code mappings for QuicErrorCodes are currently defined in [quic_protocol.h]
TODO: **hard code numbers and add them here**

0: **QUIC_NO_ERROR.** There was no error. This is not valid for RST_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 kMaxPacketSize.
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 prenegotiated (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 sequence number to be reused.

Encryption

All QUIC packets are encrypted except for the initial packets where the shared secret is being established. These packets are verified with a FNV128 hash.

The details of QUIC crypto can be found here

Priority

TODO: implement

QUIC will use the SPDY prioritization mechanism. Roughly speaking, 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.

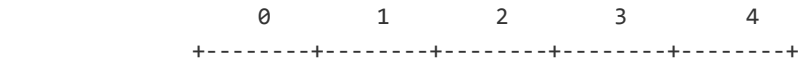
Pacing and congestion control

For the initial launch, QUIC supports two different congestion feedback algorithms. The first is an attempt to mimic TCP's cubic backoff as much as possible, given that QUIC handles out-of-order packets better than TCP. The second is an inter-arrival scheme, modeled off of WebRTC. All QUIC clients and servers are required to support the TCP cubic algorithm and are encouraged to implement the other algorithms documented in the spec.

Congestion control is negotiated as documented in the crypto spec: the client may send a CGST tag with the list of supported congestion control algorithms in-order. The server then selects the algorithm to use, and responds to the client. If no CGST tag is sent, the congestion control algorithm is assumed to be TCP cubic.

Allowed values for CGST
0: TCP Cubic
1: InterArrival:

Congestion control feedback for TCP



```
| type(1)| num lost packets|  receive window |
+--------+--------+--------+--------+--------+
```

type: The congestion control type (0 for TCP)
num lost packets: The number of packets lost over the lifetime of this connection. This may wrap for long-lived connections
receive window: The tcp receive window.

Congestion control feedback for InterArrival TODO: Update ASCII art

```
     0        1        2
+--------+--------+--------+
| type(1)| num lost packets|
+--------+--------+--------+
```

type: The congestion control type (1 for InterArrival)
num lost packets: The number of packets lost over the lifetime of this connection. This may wrap for long-lived connections
Num Packets Received: An 8 bit unsigned value specifying the number of received packets in this update.
Smallest Received Packet: The lower 48 bits of the smallest sequence number represented in this update.
Smallest Delta Time: A 64 bit unsigned value specifying the delta time from connection creation when the above packet was received.
Packet Delta: A 16 bit unsigned value specifying the sequence number delta from the smallest received. Always followed immediately by a corresponding Packet Time Delta.
Packet Time Delta: A 32 bit unsigned value specifying the time delta from smallest time when the preceding packet sequence number was received.
TODO: include a hash for included packets?
TODO: update code to kill off fields patrik doesn't need

Reliable streams over QUIC

As mentioned in the section on streams, streams are implicitly established...

Acking and retransmitting

TODO: implement and document

Stream cancellation

TODO: implement and document

SPDY Layering over QUIC

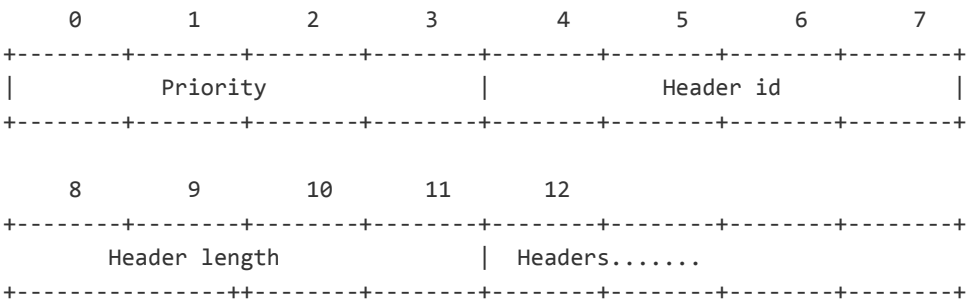
Stream management

When SPDY is sent over QUIC, the QUIC layer handles most of the stream management. SPDY stream IDs are replaced by QUIC stream IDs, and stream establishment is done by sending data on that stream. There's no explicit framing: the data sent over the QUIC stream simply consists of SPDY headers followed by the body. As with SPDY, the requests and responses are considered complete when the appropriate sender sends a frame with the FIN bit set to true. Flow control will also be handled at the QUIC layer.

Streams can be closed off if both sides set the fin flag, or via either side sending a RST_FRAME.

Stream parsing

Becasue there's no SYN stream for frames, some of the metadata in the TCP version of spdy is pre-pended to the begining of the spdy stream as follows:



Priority type: a uint32 representing the stream's priority (spdy3 style)
Header id: uint322 specifying the header id (see "compression" below)
Header length: a uint32 specifying the length of the compressed headers
Headers: spdy3-style compressed headers
Body: the body (if any) of the request

Compression

SPDY uses the gzip-style header compression used in spdy3 and below, which unfortunately means that packets with SPDY headers are head-of-line blocking because header blocks must be decompressed in the order they were compressed. The order of header blocks may not match the order of streams; server-side especially streams may responded to out-of-order (if headers for stream 5 are ready to go before headers for stream 3, for example). To make sure the headers are decompressed in the order they were compressed, a header id is included in the initial stream metadata. Both client-side server-side, the header id for the first header block compressed is 1, and will be incremented as each new header block is compressed.

The long term plan is to migrate to the possibly inaccurately named "compression" scheme which is expected to be used in spdy4. The overall plan for spdy4 is to communicate on a global, group or request level, which headers should be saved and removed. For example, on the initial request, the client would likely send a User-Agent header, with a fixed string denoting browser information. It would send that header with an opcode "store globally". Once the header frame with that opcode is acked by the server, the client doesn't ever need to send that name-value pair again: it will be implicitly added to every request. For something like cookies, the user might define a group of requests going to a specific domain and add the

name-value pair to that group. In that case, the cookie would be added to requests in that group (going to that domain) but not to all requests between client and server.

TODO: implement and either link to spdy spec, or add details.

Deployment

Unlike SPDY, one can not upgrade to QUIC as part of the SSL handshake: by the time the handshake has started, client-server interaction is already doing SSL and TCP, neither of which applies to QUIC. Instead, the alternate-protocol header is used.

To specify QUIC as an alternate protocol available on port 123, use:

```
Alternate-Protocol: 123:quic
```

When a client receives a Alternate-Protocol header advertising QUIC, it should attempt to use QUIC for future secure connections on that domain. Note that there is no guarantee that the port is open between client and server, or that intermediate servers will route packets. If the client begins communicating over QUIC and does not get responses in a timely fashion it should fail gracefully back over to the prior protocol. It should persist the failure for the browser session so that it doesn't re-attempt in the near future.

Note that the server may reply with multiple field values or a comma-separated field value for Alternate-Protocol (and in this manner indicate the various SPDY versions and 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.

To mandate https rather than QUIC for a given domain, one could send:

```
Alternate-Protocol-Required: 443:https
```

TODO: implement and document public reset packet.

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.