Network Working Group
Internet-Draft
Intended status: Informational
Expires: January 9, 2017

R. Hamilton
J. Iyengar
I. Swett
A. Wilk
Google

July 8, 2016

QUIC: A UDP-Based Multiplexed and Secure Transport draft-hamilton-quic-transport-protocol-00

Abstract

QUIC (Quick UDP-based Internet Connection) is a multiplexed and secure transport protocol that runs on top of UDP. QUIC builds on past transport experience, and implements mechanisms that make it useful as a modern general-purpose transport protocol. Using UDP as the basis of QUIC is intended to address compatibility issues with legacy clients and middleboxes. QUIC authenticates all of its headers, preventing third parties from from changing them. QUIC encrypts most of its headers, thereby limiting protocol evolution to QUIC endpoints only. Therefore, middleboxes, in large part, are not required to be updated as new protocol versions are deployed. This document describes the core QUIC protocol, including the conceptual design, wire format, and mechanisms of the QUIC protocol for connection establishment, stream multiplexing, stream and connection-level flow control, and data reliability. Accompanying documents describe QUIC's loss recovery and congestion control, and the use of TLS1.3 for key negotiation.

Table of Contents

1. Introduction

QUIC (Quick UDP-based Internet Connection) is a multiplexed and secure transport protocol that runs on top of UDP. QUIC builds on past transport experience, and implements mechanisms that make it useful as a modern general-purpose transport protocol. Using UDP as the basis of QUIC is intended to address compatibility issues with legacy clients and middleboxes. QUIC authenticates all of its headers, preventing third parties from from changing them. QUIC encrypts most of its headers, thereby limiting protocol evolution to QUIC endpoints only. Therefore, middleboxes, in large part, are not required to be updated as new protocol versions are deployed.

This document describes the core QUIC protocol, including the conceptual design, wire format, and mechanisms of the QUIC protocol

for connection establishment, stream multiplexing, stream and connection-level flow control, and data reliability. Accompanying documents describe QUIC's loss recovery and congestion control [draft-iyengar-quic-loss-recovery], and the use of TLS1.3 for key negotiation [draft-thomson-quic-tls].

2. Conventions and Definitions

Definitions of terms that are used in this document:

- "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 logical, bi-directional channel of ordered bytes 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.

3. A QUIC Overview

We now briefly describe QUIC's key mechanisms and benefits. Key strengths of QUIC include:

- Low-latency Version Negotiation
- Low-latency connection establishment
- Multiplexing without head-of-line blocking
- Authenticated and encrypted header and payload
- Rich signaling for congestion control and loss recovery
- Stream and connection flow control
- Resilience to NAT rebinding

3.1. Low-Latency Version Negotiation

(TODO: add text here)

3.2. Low-Latency Connection Establishment

QUIC relies on a combined crypto and transport handshake for setting up a secure transport connection. QUIC connections are expected to commonly use 0-RTT handshakes, meaning that for most QUIC connections, data can be sent immediately following the client handshake packet, without waiting for a reply from the server. QUIC provides a dedicated stream (Stream ID 1) to be used for performing the crypto handshake and QUIC options negotiation. Theformat of the QUIC options and parameters used during negotiation are described in this document, but the handshake protocol that runs on Stream ID 1 is described in the accompanying crypto handshake draft [draft-thomson-quic-tls].

3.3. Stream Multiplexing

When application messages are transported over TCP, independent application messages can suffer from head-of-line blocking. When an application 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 application streams that are encapsulated in subsequent segments.

Because it is designed from the ground up for multiplexed operation, QUIC ensures that 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.

3.4. Rich Signaling for Congestion Control and Loss Recovery

QUIC's packet framing and acknowledgments carry rich information that help both congestion control and loss recovery in fundamental ways. Each QUIC packet, both original and retransmitted, carries a new packet number. This allows a QUIC sender to distinguish ACKs for retransmissions from ACKs for original transmissions, thus avoiding TCP's retransmission ambiguity problem. QUIC acknowledgments also explicitly encode 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. QUIC's ACK frames support up to 256 ack blocks, so QUIC is more resilient to reordering than TCP with SACK support, as well as able to keep more bytes on the wire when there is

reordering or loss. A QUIC endpoint has an accurate picture of which packets the peer has received.

3.5. Stream and Connection Flow Control

QUIC implements stream- and connection-level flow control, closely following HTTP/2's flow control mechanisms. At a high level, QUIC's stream-level flow control works as follows. A QUIC receiver advertises the absolute byte offset within each stream up to 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 all streams on 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.

3.6. Authenticated and Encrypted Header and Payload

TCP headers appear in plaintext on the wire and are 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 to improve TCP performance. However, even "performance-enhancing" middleboxes 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.

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

3.7. Resilience to NAT Rebinding

QUIC connections are identified by a 64-bit Connection ID, randomly generated by the client. QUIC's consistent connection ID allows connections to survive changes to the client's IP and/or port, such as those caused by NAT rebindings. QUIC also provides automatic cryptographic verification of a rebound client, since the client

continues to use the same session key for encrypting and decrypting packets.

4. Connection Establishment

QUIC's connection establishment intertwines version negotiation with the crypto and transport handshakes to reduce connection establishment latency. We first describe version negotiation, since the subsequent crypto and transport handshakes rely on successful version negotiation.

4.1. Version Negotiation

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 a 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 MUST compare the client's version to the versions it supports.

- If the client's version is acceptable to the server, the server MUST use this protocol version for the lifetime of the connection. All subsequent packets sent by the server MUST have the version flag off.

- If the client's version is not acceptable to the server, the server MUST 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. In this case, a 1-RTT delay is incurred before the server can process client data.

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 packet with the version flag off 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 client and server must include their supported versions in their corresponding 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.

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

4.2. Combined Crypto and Transport Handshake

QUIC relies on a combined crypto and transport handshake to minimize connection establishment latency. QUIC provides a dedicated stream (Stream ID 1) to be used for performing a combined connection and security handshake. The crypto handshake protocol encapsulates and delivers QUIC's transport handshake to the peer on the crypto stream; the first QUIC packet from the client to the server MUST carry handshake information as a Stream Frame on the crypto stream.

4.2.1. Transport Parameters and Options

The transport component of the handshake is responsible for exchanging and/or negotiating the following parameters for a QUIC connection. Not all parameters are negotiated, some are parameters sent in just one direction. These parameters and options are encoded and handed off to the crypto handshake protocol to be transmitted to the peer.

4.2.1.1. Encoding

(TODO: Describe format with example)

QUIC encodes the transport parameters and options as tag-value pairs, all as 7-bit ASCII strings. QUIC parameter tags are listed below.

4.2.1.2. Required Transport Parameters

- SFCW: Stream Flow Control Window. The stream level flow control byte offset advertised by the sender of this parameter.

- CFCW: Connection Flow Control Window. The connection level flow control byte offset advertised by the sender of this parameter.

- MSPC: Maximum number of incoming streams per connection.

4.2.1.3. Optional Transport Parameters

- 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 the common case when an ephemeral UDP port is used for a single QUIC 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. (TODO: List connection options.)

4.2.2. Proof of Source Address Ownership

Transport protocols commonly use a roundtrip time to verify a client's address ownership for protection from malicious clients that spoof their source address. QUIC uses a cookie, called the Source Address Token (STK), to mostly eliminate this roundtrip of delay. This technique is similar to TCP Fast Open's use of a cookie to avoid a roundtrip of delay in TCP connection establishment.

On a new connection, a QUIC server sends an STK, which is opaque to and stored by the client. On a subsequent connection, the client echoes it in the transport handshake as proof of IP ownership.

A QUIC server also uses the STK to store server-designated connection IDs for Stateless Rejects, to verify that an incoming connection contains the correct connection ID.

A QUIC server MAY additionally store other data in a the STK, such as measured bandwidth and measured minimum RTT to the client that may help the server better bootstrap a subsequent connection from the same client. A server MAY send an updated STK message mid-connection to update server state that is stored at the client in the STK.

(TODO: Describe server and client actions on STK, encoding, recommendations for what to put in an STK. Describe SCUP messages.)

4.2.3. Crypto Handshake Protocol Features

QUIC does not restrict itself to using a specific handshake protocol, so the details of a specific handshake protocol are out of this document's scope. If not explicitly specified in the application mapping, TLS is assumed to be the default crypto handshake protocol, as described in [draft-mthomson-quic-tls]. An application that maps

to QUIC MAY however specify an alternative crypto handshake protocol to be used.

The following list of requirements and recommendations documents properties of the current prototype handshake which should be provided by any handshake protocol.

- Transport Negotiation: The crypto handshake MUST provide a mechanism for the transport component to exchange transport parameters and Source Address Tokens.

- Connection Establishment in 0-RTT: Since low-latency connection establishment is a critical feature of QUIC, the QUIC handshake protocol SHOULD attempt to achieve 0-RTT connection establishment latency for repeated connections between the same endpoints.

- Source Address Spoofing Defense: Since QUIC handles source address verification, the crypto protocol SHOULD NOT impose a separate source address verification mechanism.

- Server Config Update: A QUIC server may refresh the source-address token (STK) mid-connection, to update the information stored in the STK at the client and to extend the period over which 0-RTT connections can be established by the client. A crypto protocol

- Certificate Compression: Early QUIC experience demonstrated that compressing certificates exchanged during a handshake is valuable in reducing latency. This additionally helps to reduce the amplification attack footprint when a server sends a large set of certificates, which is not uncommon with TLS. The crypto protocol SHOULD compress certificates and any other information to minimize the number of packets sent during a handshake.

5. Streams: QUIC's Data Structuring Abstraction

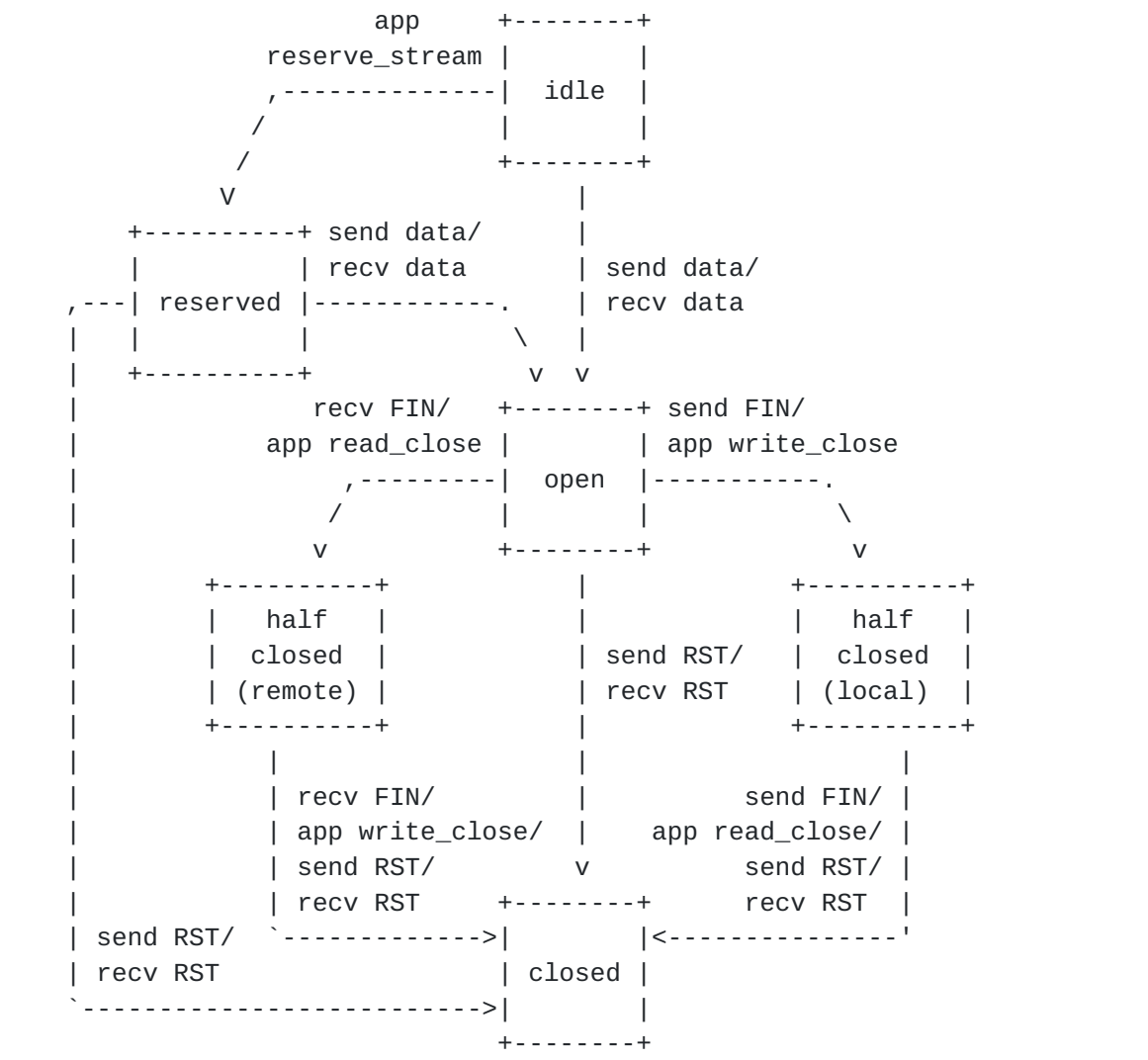
Streams in QUIC provide a lightweight, ordered, and bidirectional byte-stream abstraction. Streams can be created either by the client or the server, can concurrently send data interleaved with other streams, and can be cancelled. QUIC's stream lifetime is modeled closely after HTTP/2's [RFC7540]. Streams are independent of each other in delivery order. That is, data that is received on a stream is delivered in order within that stream, but there is no particular delivery order across streams. Transmit ordering among streams is left to the implementation. (TODO: Perhaps define HTTP/2-like priority scheme, including a PRIORITY frame for QUIC stream priorities.) QUIC streams are considered lightweight in that the creation and destruction of streams are expected to have minimal bandwidth and computational cost. A single STREAM frame may create,

carry data for, and terminate a stream, or a stream may last the entire duration of a connection. Implementations are therefore advised to keep these extremes in mind and to implement stream creation and destruction to be as lightweight as possible.

An alternative view of QUIC streams is as an elastic "message" abstraction, similar to the way ephemeral streams are used in SST [cite SST], which may be a more appealing description for some applications.

5.1. Life of a Stream

The semantics of QUIC streams is based on HTTP/2 streams, and the lifecycle of a QUIC stream therefore closely follows that of an HTTP/2 stream [RFC7540], with some differences to accommodate the possibility of out-of-order delivery in QUIC. The lifecycle of a QUIC stream is shown in the following figure and described below.



send: endpoint sends this frame
recv: endpoint receives this frame

data: application data in a STREAM frame
FIN: FIN flag in a STREAM frame
RST: RST_STREAM frame

app: application API signals to QUIC
reserve_stream: causes a StreamID to be reserved for later use
read_close: causes stream to be half-closed without receiving a FIN
write_close: causes stream to be half-closed without sending a FIN

Note that this diagram shows stream state transitions and the frames and flags that affect those transitions only. For the purpose of state transitions, the FIN flag is processed as a separate event to the frame that bears it; a STREAM frame with the FIN flag set can cause two state transitions. When the FIN bit is sent on an empty

STREAM frame, the offset in the STREAM frame MUST be one greater than the last data byte sent on this stream.

Both endpoints have a subjective view of the state of a stream that could be different when frames are in transit. Endpoints do not coordinate the creation of streams; they are created unilaterally by either endpoint. The negative consequences of a mismatch in states are limited to the "closed" state after sending RST_STREAM, where frames might be received for some time after closing.

Streams have the following states:

5.1.1. idle

All streams start in the "idle" state.

The following transitions are valid from this state:

Sending or receiving a STREAM frame causes the stream to become "open". The stream identifier is selected as described in Section XX. The same STREAM frame can also cause a stream to immediately become "half-closed".

An application can reserve an idle stream for later use. The stream state for the reserved stream transitions to "reserved".

Receiving any frame other than STREAM or RST_STREAM on a stream in this state MUST be treated as a connection error (Section XX) of type YYYY.

5.1.2. reserved

A stream in this state has been reserved for later use by the application. In this state only the following transitions are possible:

- Sending or receiving a STREAM frame causes the stream to become "open".

- Sending or receiving a RST_STREAM frame causes the stream to become "closed".

5.1.3. open

A stream in the "open" state may be used by both peers to send frames of any type. In this state, a sending peer must observe the flow-control limit advertised by its receiving peer (Section XX).

From this state, either endpoint can send a frame with the FIN flag set, which causes the stream to transition into one of the "half-closed" states. An endpoint sending an FIN flag causes the stream state to become "half-closed (local)". An endpoint receiving a FIN flag causes the stream state to become "half-closed (remote)"; the receiving endpoint MUST NOT process the FIN flag until all preceding data on the stream has been received.

Either endpoint can send a RST_STREAM frame from this state, causing it to transition immediately to "closed".

5.1.4. half-closed (local)

A stream that is in the "half-closed (local)" state MUST NOT be used for sending STREAM frames; WINDOW_UPDATE and RST_STREAM MAY be sent in this state.

A stream transitions from this state to "closed" when a frame that contains an FIN flag is received or when either peer sends a RST_STREAM frame.

An endpoint can receive any type of frame in this state. Providing flow-control credit using WINDOW_UPDATE frames is necessary to continue receiving flow-controlled frames. In this state, a receiver MAY ignore WINDOW_UPDATE frames for this stream, which might arrive for a short period after a frame bearing the FIN flag is sent.

5.1.5. half-closed (remote)

A stream that is "half-closed (remote)" is no longer being used by the peer to send any data. In this state, a sender is no longer obligated to maintain a receiver stream-level flow-control window.

If an endpoint receives any STREAM frames for a stream that is in this state, it MUST close the connection with a QUIC_STREAM_DATA_AFTER_TERMINATION error (Section XX).

A stream in this state can be used by the endpoint to send frames of any type. In this state, the endpoint continues to observe advertised stream-level and connection-level flow-control limits (Section XX).

A stream can transition from this state to "closed" by sending a frame that contains a FIN flag or when either peer sends a RST_STREAM frame.

5.1.6. closed

The "closed" state is the terminal state.

A final offset is present in both a frame bearing a FIN flag and in a RST_STREAM frame. Upon sending either of these frames for a stream, the endpoint MUST NOT send a STREAM frame carrying data beyond the final offset.

An endpoint that receives any frame for this stream after receiving either a FIN flag and all stream data preceding it, or a RST_STREAM frame, MUST quietly discard the frame, with one exception. If a STREAM frame carrying data beyond the received final offset is received, the endpoint MUST close the connection with a QUIC_STREAM_DATA_AFTER_TERMINATION error (Section XX).

An endpoint that receives a RST_STREAM frame (and which has not sent a FIN or a RST_STREAM) MUST immediately respond with a RST_STREAM frame, and MUST NOT send any more data on the stream. This endpoint may continue receiving frames for the stream on which a RST_STREAM is received.

If this state is reached as a result of sending a RST_STREAM frame, the peer that receives the RST_STREAM might have already sent -- or enqueued for sending -- frames on the stream that cannot be withdrawn. An endpoint MUST ignore frames that it receives on closed streams after it has sent a RST_STREAM frame. An endpoint MAY choose to limit the period over which it ignores frames and treat frames that arrive after this time as being in error.

STREAM frames received after sending RST_STREAM are counted toward the connection and stream flow-control windows. Even though these frames might be ignored, because they are sent before their sender receives the RST_STREAM, the sender will consider the frames to count against its flow-control windows.

In the absence of more specific guidance elsewhere in this document, implementations SHOULD treat the receipt of a frame that is not expressly permitted in the description of a state as a connection error (Section XX). Frames of unknown types are ignored.

(TODO: QUIC_STREAM_NO_ERROR is a special case. Write it up.)

5.2. Stream Identifiers

Streams are identified by an unsigned 32-bit integer, referred to as the StreamID. To avoid StreamID collision, clients MUST initiate

streams usinge odd-numbered StreamIDs; streams initiated by the server MUST use even-numbered StreamIDs.

A StreamID of zero (0x0) is reserved and used for connection-level flow control frames (Section XX); the StreamID of zero cannot be used to establish a new stream.

StreamID 1 (0x1) is reserved for the crypto handshake. StreamID 1 MUST NOT be used for application data, and MUST be the first client-initiated stream.

Streams MAY be created in arbitrary order. A QUIC endpoint MUST NOT reuse a StreamID on a given connection.

5.3. Stream Concurrency

An endpoint can limit the number of concurrently active incoming streams by setting the MSPC parameter (see Section XX) in the transport parameters. The maximum concurrent streams setting is specific to each endpoint and applies only to the peer that receives the setting. That is, clients specify the maximum number of concurrent streams the server can initiate, and servers specify the maximum number of concurrent streams the client can initiate.

Streams that are in the "open" state or in either of the "half-closed" states count toward the maximum number of streams that an endpoint is permitted to open. Streams in any of these three states count toward the limit advertised in the MSPC setting.

Endpoints MUST NOT exceed the limit set by their peer. An endpoint that receives a STREAM frame that causes its advertised concurrent stream limit to be exceeded MUST treat this as a stream error of type QUIC_TOO_MANY_OPEN_STREAMS (Section XX).

5.4. Sending and Receiving Data

Once a stream is created, endpoints may use the stream to send and receive data. Each endpoint may send a series of STREAM frames encapsulating data on a stream until the stream is terminated in that direction. Streams are an ordered byte-stream abstraction, and they have no other structure within them. STREAM frame boundaries are not expected to be preserved in retransmissions from the sender or during delivery to the application at the receiver.

When new data is to be sent on a stream, a sender MUST set the encapsulating STREAM frame's offset field to the stream offset of the first byte of this new data. A receiver MUST ensure that received stream data is delivered to the application as an ordered byte-

stream. Data received out of order MUST be buffered for later delivery, as long as it is not in violation of the receiver's flow control limits.

An endpoint MUST NOT send any stream data without consulting the congestion controller and the flow windows, with one exception in the case of connection-level flow control, as described in Section XX. The congestion controller is described in the companion document [draft-loss-recovery].

6. Packetization and Reliability

The maximum packet size for QUIC is the maximum size of the encrypted payload of the resulting UDP datagram. A default maximum packet size of 1350 bytes is recommended. Endpoints SHOULD use PLPMTUD [RFC4821] for detecting the path's MTU and setting the maximum packet size appropriately.

A sender bundles one or more frames to send in a Regular QUIC Packet. A sender MAY bundle any set of frames in a packet. All QUIC Packets MUST contain a Packet Sequence Number (PSN) and MAY contain one or more frames (Section XX). PSNs MUST be unique within a connection and MUST NOT be reused within the same connection. PSNs MUST be assigned to packets in a strictly monotonically increasing order.

A sender SHOULD minimize per-packet bandwidth and computational costs by bundling as many frames as possible within a QUIC packet. A sender MAY wait for a short period of time to bundle multiple frames before sending a packet that is not maximally packed, to avoid sending out large numbers of small packets.

Regular QUIC Packets are "containers" of frames; a packet is never retransmitted whole, but frames in a lost packet may be rebundled and transmitted in a subsequent packet as necessary.

A packet may contain frames and/or application data, only some of which may require reliability. When a packet is detected as lost, the sender SHOULD only rebundle frames and application data that require retransmission.

- All application data sent in STREAM frames MUST be retransmitted, with one exception. When an endpoint sends a RST_STREAM frame, data outstanding on that stream SHOULD NOT be retransmitted, since subsequent data on this stream is expected to not be delivered by the receiver.

- ACK, STOP_WAITING, and PADDING frames MUST NOT be retransmitted. New frames of these types may however be bundled with any outgoing packet.

- All other frames MUST be retransmitted.

Upon detecting losses, a sender MUST take appropriate congestion control action. The details of loss detection and congestion control are described in [draft-loss-recovery].

A receiver acknowledges receipt of a received packet by sending one or more ACK frames containing the PSN of the received packet. To avoid perpetual acking between endpoints, a receiver MUST NOT generate an ack in response to every packet containing only ACK frames (TODO: Describe acking acks.) Strategies and implications of the frequency of generating acknowledgments are discussed in more detail in [draft-loss-recovery].

7. Flow Control

QUIC employs a credit-based flow-control scheme similar to HTTP/2's flow control [RFC7540].

A receiver advertises the number of octets it is prepared to receive on a given stream and for the entire connection. This leads to two levels of flow control in QUIC: (i) Connection flow control, which prevents senders from exceeding a receiver's buffer capacity for the connection, and (ii) Stream flow control, which prevents a single stream from consuming the entire receive buffer for a connection.

A receiver sends WINDOW_UPDATE frames to the sender to advertise additional credit, for both connection and stream flow control. A receiver advertisesadvertise the maximum absolute byte offset in the stream or in the connection which the receiver is willing to receive.

The initial flow control credit is 65536 bytes for both the stream and connection flow controllers.

A receiver MAY advertise a larger offset at any point in the connection by sending a WINDOW_UPDATE frame. A receiver MUST NOT renege on an advertisement; that is, once a receiver advertises an offset via a WINDOW_UPDATE frame, it MUST NOT subsequently advertise a smaller offset. A sender may receive WINDOW_UPDATE frames out of order; a sender MUST therefore ignore any reductions in flow control credit.

A sender MUST send BLOCKED frames to indicate it has data to write but is blocked by lack of connection or stream flow control credit.

BLOCKED frames are expected to be sent infrequently in common cases, but they MAY be useful for debugging and monitoring purposes.

A receiver advertises credit for a stream by sending a WINDOW_UPDATE frame with the StreamID set appropriately. A receiver may simply use the current received offset to determine the flow control offset to be advertised.

Connection flow control is a limit to the total bytes of stream data sent in STREAM frames. A receiver advertises credit for a connection by sending a WINDOW_UPDATE frame with the StreamID set to zero (0x00). A receiver may maintain a cumulative sum of bytes received cumulatively on all streams to determine the value of the connection flow control offset to be advertised in WINDOW_UPDATE frames. A sender may maintain a cumulative sum of stream data bytes sent to impose the connection flow control limit.

7.1. Important considerations

There are some edge cases which must be considered when dealing with stream and connection level flow control. Given enough time, both endpoints must agree on flow control state. If one end believes it can send more than the other end is willing to receive, the connection will be torn down when too much data arrives. Conversely if a sender believes it is blocked, while endpoint B expects more data can be received, then the connection can be in a deadlock, with the sender waiting for a WINDOW_UPDATE which will never come.

7.1.1. Mid-stream RST_STREAM

On receipt of an RST_STREAM frame, an endpoint will tear down state for the matching stream and ignore further data arriving on that stream. This could result in the endpoints getting out of sync, since the RST_STREAM frame may have arrived out of order and there may be further bytes in flight. The data sender would have counted the data against its connection level flow control budget, but a receiver that has not received these bytes would not know to include them as well. The receiver must learn of the number of bytes that were sent on the stream to make the same adjustment in its connection flow controller.

To avoid this de-synchronization, a RST_STREAM sender MUST include the final byte offset sent on the stream in the RST_STREAM frame. On receiving a RST_STREAM frame, a receiver definitively knows how many bytes were sent on that stream before the RST_STREAM frame, and the receiver MUST use the final offset to account for all bytes sent on the stream in its connection level flow controller.

7.1.2. Response to a RST_STREAM

Since streams are bidirectional, a sender of a RST_STREAM needs to know how many bytes the peer has sent on the stream. If an endpoint receives a RST_STREAM frame and has sent neither a FIN nor a RST_STREAM, it MUST send a RST_STREAM in response, bearing the offset of the last byte sent on this stream as the final offset.

7.1.3. Offset Increment

This document leaves when and how many bytes to advertise in a WINDOW_UPDATE to the implementation, but offers a few considerations. WINDOW_UPDATE frames constitute overhead, and therefore, sending a WINDOW_UPDATE with small offset increments is undesirable. At the same time, sending WINDOW_UPDATES with large offset increments requires the sender to commit to that amount of buffer. Implementations must find the correct tradeoff between these sides to determine how large an offset increment to send in a WINDOW_UPDATE.

A receiver MAY use an autotuning mechanism to tune the size of the offset increment to advertise based on a roundtrip time estimate and the rate at which the receiving application consumes data, similar to common TCP implementations.

7.1.4. BLOCKED frames

If a sender does not receive a WINDOW_UPDATE frame when it has run out of flow control credit, the sender will be blocked and may send a BLOCKED frame. A receiver should not wait for a BLOCKED frame before responding with a WINDOW_UPDATE, since doing so will cause at least one roundtrip of quiescence. Further, if blocked, a sender will go into quiescence, which may result in poor performance of the congestion controller. For smooth operation of the congestion controller, it is generally considered best to not let the sender go into quiescence if avoidable. To avoid blocking a sender, and reasonably accounting for the possibiity of loss, a receiver should send a WINDOW_UPDATE frame at least two roundtrip times before the sender gets blocked.

8. Connection Termination

Connections should remain open until they become idle for a pre-negotiated period of time. 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 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.

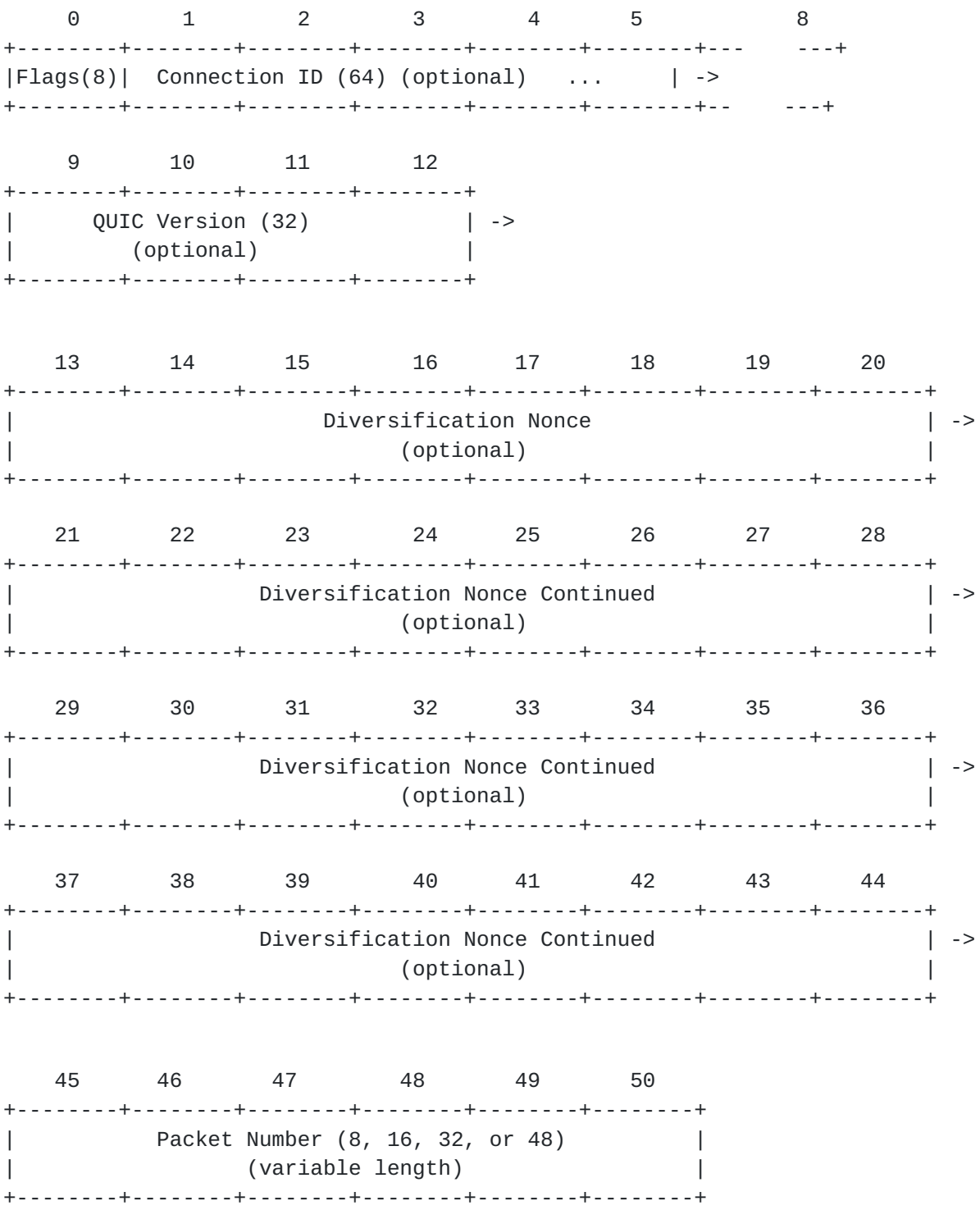
An endpoint may also send a PUBLIC_RESET packet at any time during the connection to abruptly terminate an active connection.

9. Packet Types and Formats

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

9.1. Public Packet Header

All QUIC packets on the wire begin with a public header sized between 2 and 19 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:

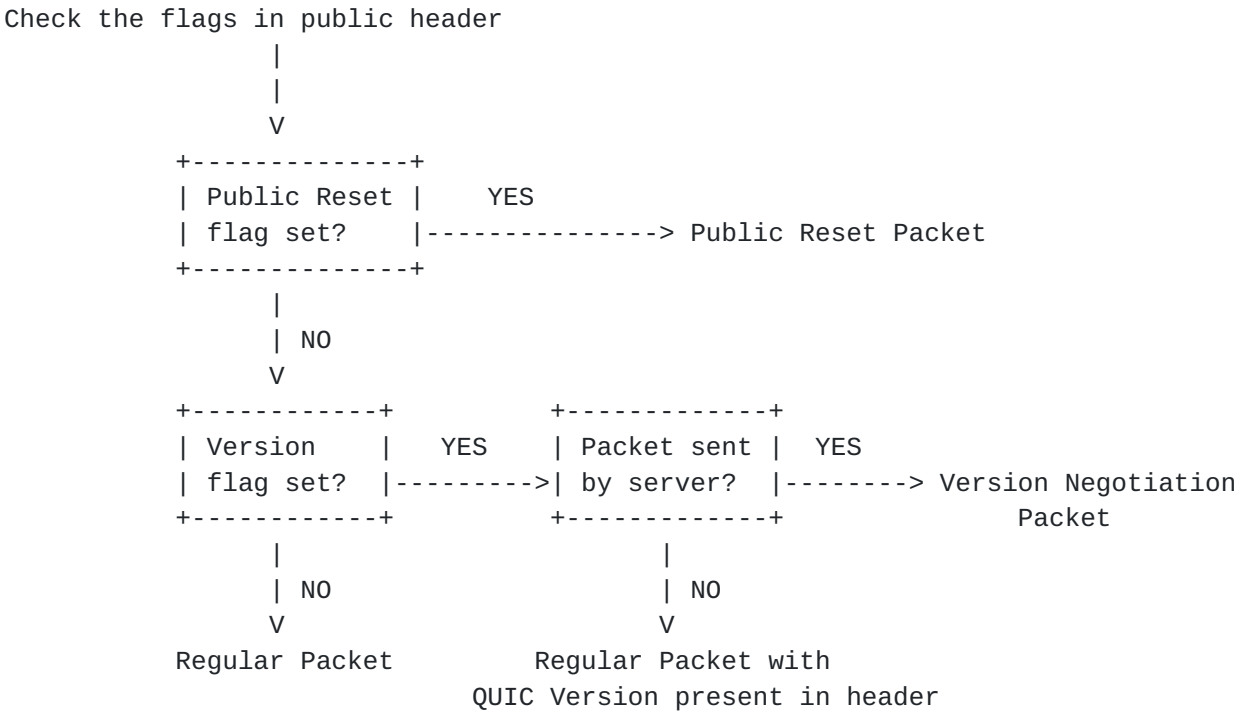
- Flags:
  - 0x01 = FLAG_VERSION. The semantics of this flag depends on whether the packet is sent by the server or the client. A client sets this flag to indicate that the header contains a QUIC version (see below). A client MUST set this bit in all packets until confirmation from the server arrives agreeing to the proposed version. A server indicates agreement on a version by sending packets without setting this bit. When the server sets this bit, the packet is a Version Negotiation Packet. Version Negotiation is described in more detail later.
  - 0x02 = FLAG_PUBLIC_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 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 indicates the 5-tuple fully identifies the connection, so connection is is optional).
  - 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 flags contain FLAG_VERSION (i.e 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 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 received from a peer shall be inferred to have the value closest to the one more than the largest known packet number received from that peer. The transmitted portion of the packet number matches the lowest bits of the inferred value.

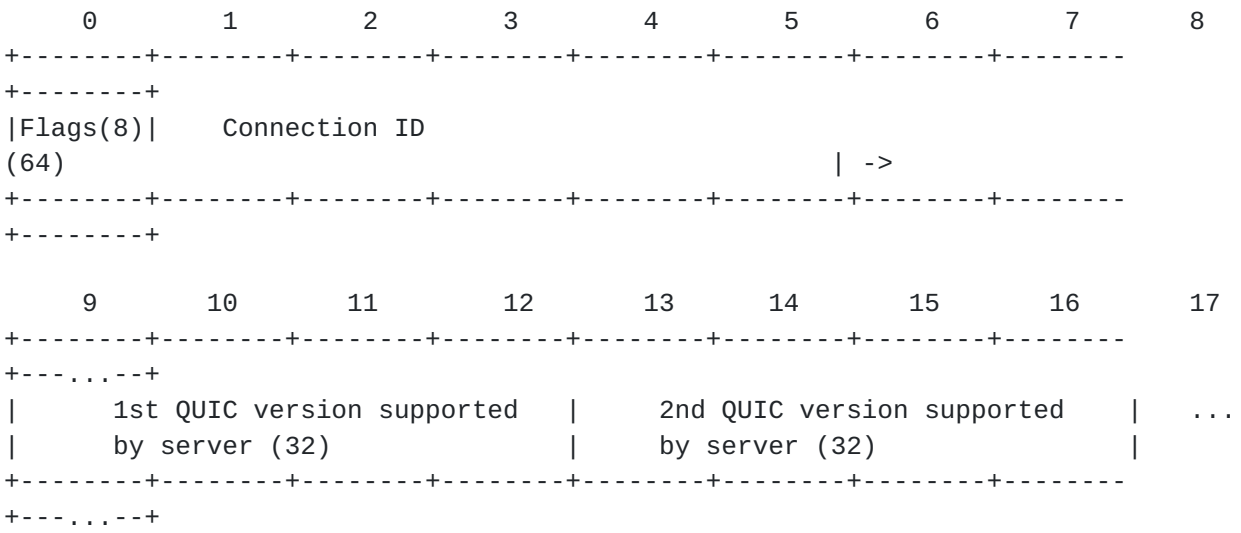
A Flags processing flowchart follows:



9.2. Special Packets

9.2.1. Version Negotiation Packet

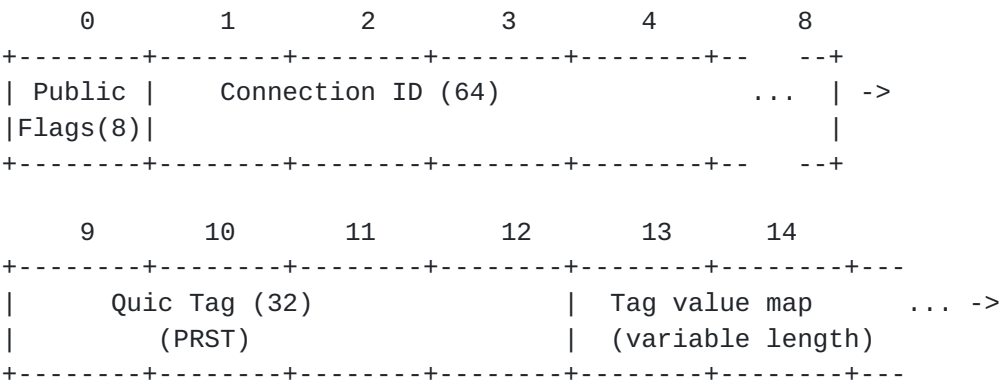
A Version Negotiation packet is only sent by the server. Version Negotiation packets begin with an 8-bit flags and 64-bit Connection ID. The flags must set FLAG_VERSION and indicate the 64-bit Connection ID. The rest of the Version Negotiation packet is a list of 4-byte versions which the server supports:



9.2.2. Public Reset Packet

A Public Reset packet begins with an 8-bit flags and 64-bit Connection ID. The PUBLIC_FLAG_RESET flag MUST be set and the header MUST indicate the entire 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 ():



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.

9.3. Regular Packets

Each Regular Packet consists of a Public Header followed by a series of data frames. The Public Header is authenticated but not encrypted, and the rest of the packet starting with the first frame both authenticated and 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, as described in the following section.

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

10. Frame Types and Formats

A single Regular Packet MAY contain multiple frames and multiple frame types. Frames MUST fit within a single QUIC Packet and MUST NOT span a QUIC Packet boundary. Each Frame begins with a Frame Type byte, indicating its type, followed by type-dependent header fields, and variable-length data.

10.1. Frames

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:

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

10.2. Frame Types

There are two types of Frames: 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:

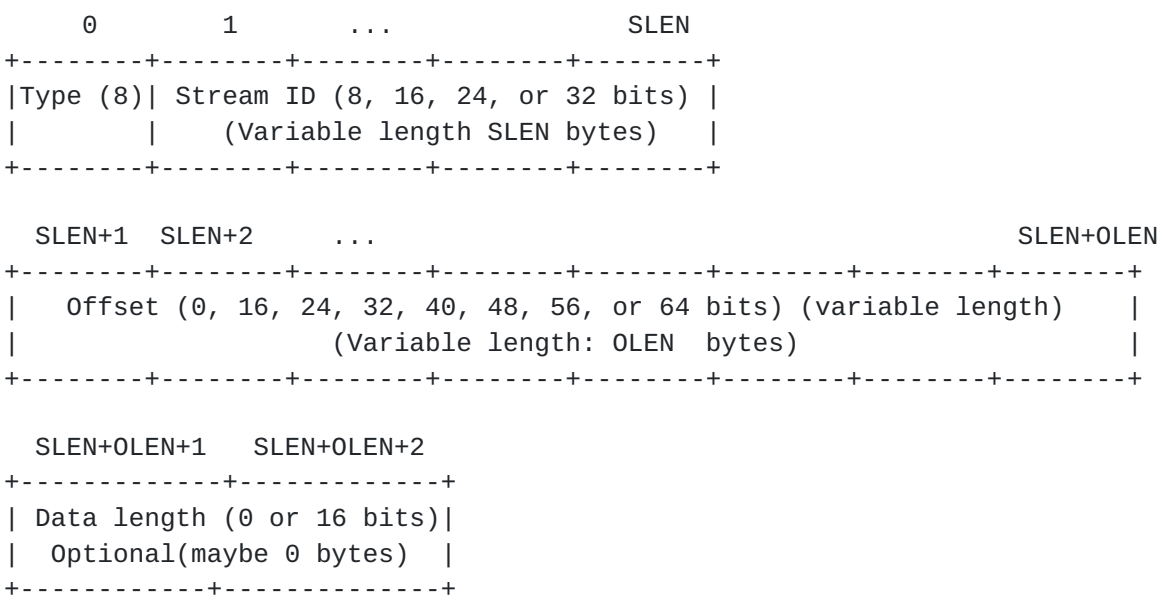
Type-field value	Control Frame-type
1fdooossB	STREAM
01ntllmmB	ACK

Currently defined Regular Frame Types are:

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

10.3. STREAM Frame

STREAM frames implicitly create a stream and carry stream data. A STREAM frame is shown below.



The STREAM frame header fields 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. The first byte in the stream has an offset of 0.

- 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 have either non-zero data length or the FIN bit set.

Stream multiplexing is achieved by interleaving STREAM frames from multiple streams into one or more QUIC Packets. A single QUIC packet MAY bundle STREAM frames from multiple streams.

Implementation note: One of the benefits of QUIC is avoidance of head-of-line blocking across multiple streams. When a packet loss occurs, only streams with data in that packet are blocked waiting for a retransmission to be received, while other streams can continue making progress. Note that when data from multiple streams is bundled into a single QUIC packet, loss of that packet blocks all those streams from making progress. An implementation is therefore advised to bundle as few streams as necessary in outgoing packets without losing transmission efficiency to underfilled packets.

10.4. ACK Frame

Receivers send ACK frames to inform senders which packets they have received, as well as which packets it still considers missing. 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 sender, the sender 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. Once a packet is acked, even if it does not appear in a future ack frame, it is assumed to be acked.

A sender MAY intentionally skip packet numbers to introduce entropy into the connection, to avoid opportunistic ack attacks. The sender MUST close the connection if an unsent packet number is acked. The

format of the ACK frame is efficient at expressing blocks of missing packets; skipping packet sequence numbers between 1 and 255 effectively provides up to 8 bits of efficient entropy on demand, which should be adequate protection against most opportunistic ack attacks.

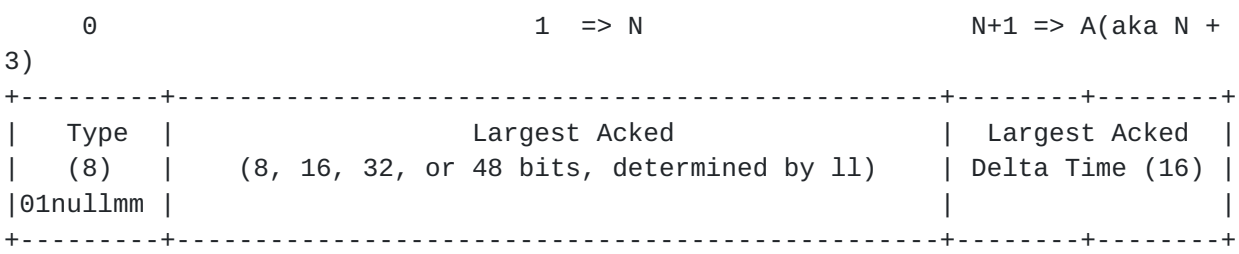
Section Offsets

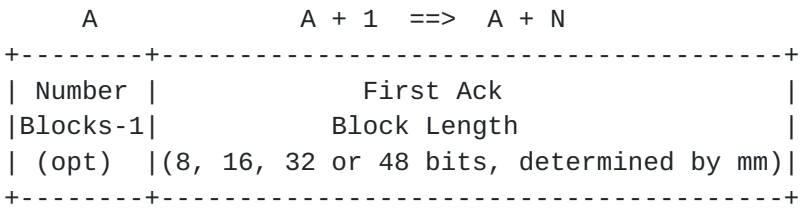
0: Start of the ack frame.

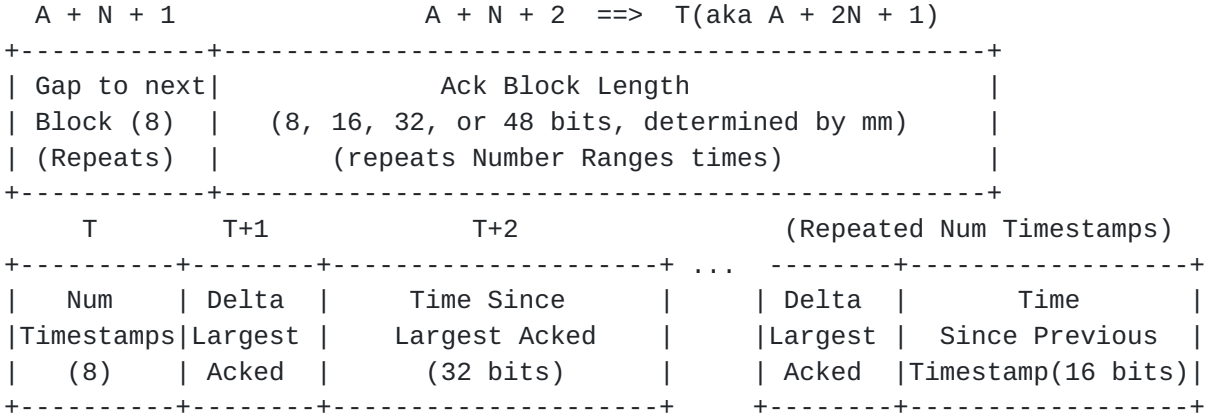
T: Byte offset of the start of the timestamp section.

A: Byte offset of the start of the ack block section.

N: Length in bytes of the largest acked.







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.

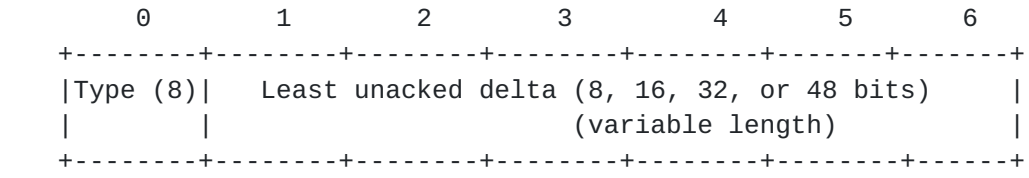
    * Ack block length: A variable-sized packet number delta. For the first missing packet range, the ack block starts at largest acked. For the first ack block, the length of the ack block is 1 + this value. For subsequent ack blocks, it is the length of the ack block. For non-first blocks, a value of 0 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.

10.5. 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 Flags field.) The frame is as follows:



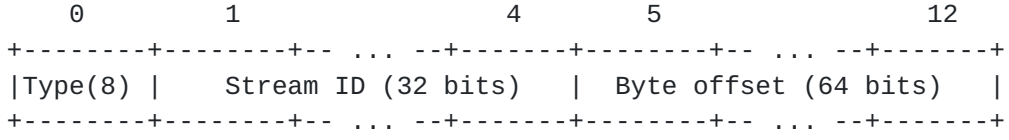
The fields in the STOP_WAITING frame are as follows:

- 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.

10.6. WINDOW_UPDATE Frame

The WINDOW_UPDATE frame informs the peer of an increase in an endpoint's flow control receive window. The StreamID can be zero, indicating this WINDOW_UPDATE applies to the connection level flow control window, or non-zero, indicating that the specified stream should increase its flow control window. The frame is as follows:



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.

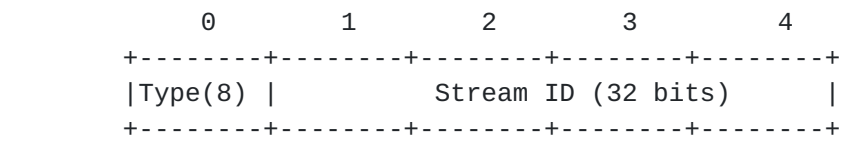
An absolute byte offset is specified, and the receiver of a WINDOW_UPDATE frame may only send up to that number of bytes on the specified stream. Violating flow control by sending further bytes will result in the receiving endpoint closing the connection.

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.

10.7. BLOCKED Frame

A sender sends a BLOCKED frame when it is ready to send data (and has data to send), but is currently flow control blocked. BLOCKED frames are purely informational frames, but 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:



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.

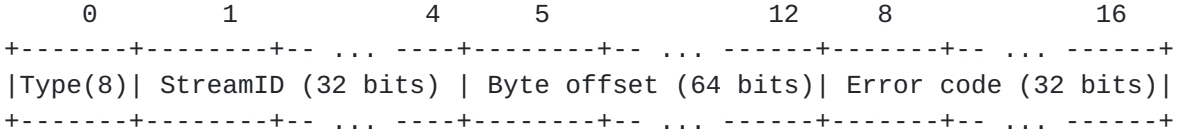
- Stream ID: A 32-bit unsigned number indicating the stream which is flow control blocked. A non-zero Stream ID field specifies the stream that is flow control blocked. When zero, the Stream ID field indicates that the connection is flow control blocked at the connection level.

10.8. 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.

10.9. RST_STREAM Frame

An endpoint may use a RST_STREAM frame to abruptly terminate a stream. The frame is as follows:



The fields are:

- 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 written on this stream by the RST_STREAM sender.

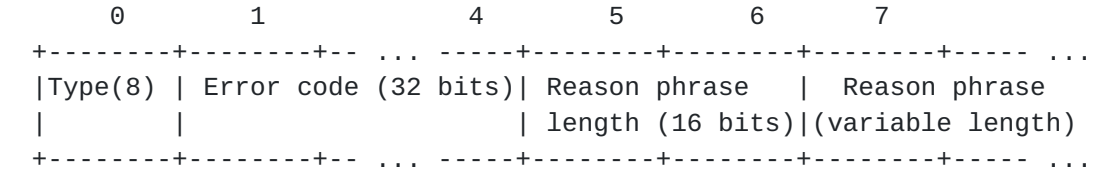
- Error code: A 32-bit error code which indicates why the stream is being closed.

10.10. PING frame

Endpoints can use PING frames to verify that their peers are still alive or to check reachability to the peer. 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 send a PING frame after 15 seconds of quiescence. A PING frame only has a Frame Type field, and must have the 8-bit Frame Type field set to 0x07.

10.11. CONNECTION_CLOSE frame

An endpoint sends a CONNECTION_CLOSE frame to notify its peer that the connection is being closed. If there are open streams that haven't been explicitly closed, they are 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.

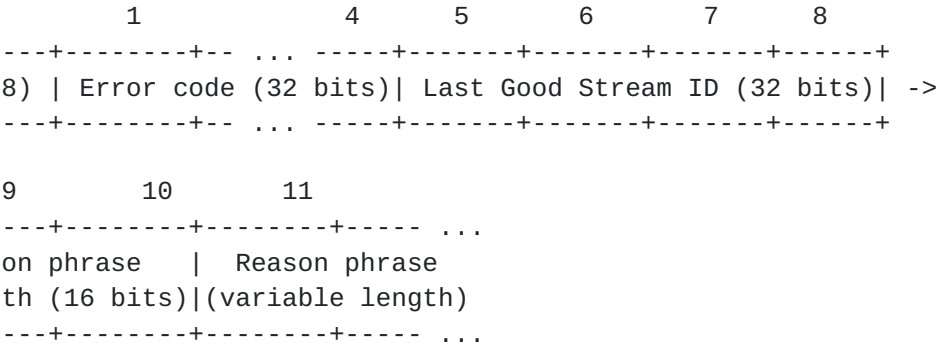
- Error Code: A 32-bit error code which indicates the reason for closing this connection.

- 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.

10.12. GOAWAY Frame

An endpoint may use a GOAWAY frame to notify its peer that the connection should stop being used, and will likely be aborted in the future. The endpoints will continue using any active streams, 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 0x06 specifying that this is a GOAWAY frame.

- Error Code: A 32-bit field error code 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.

11. Error Codes

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_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.

12. Security and Privacy Considerations

(TODO: List considerations)

13. 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 [1]. 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.

14. 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.

.

15. References

15.1. Normative References

[RFC2119] Bradner, S., "Key Words for use in RFCs to Indicate Requirement Levels", March 1997.

[draft-thomson-quic-tls] Thomson, M. and R. Hamilton, "Porting QUIC to TLS", March 2016.

[draft-iyengar-quic-loss-recovery] Iyengar, J. and I. Swett, "QUIC Loss Recovery and Congestion Control", July 2016.

15.2. Informative References

[RFC7540] Belshe, M., Peon, R., and M. Thomson, "Hypertext Transfer Protocol Version 2 (HTTP/2)", May 2015.

15.3. URIs

[1] https://goo.gl/dMVtFi

Authors' Addresses

Ryan Hamilton
Google

Email: rch@google.com

Janardhan Iyengar
Google

Email: jri@google.com

Ian Swett
Google

Email: ianswett@google.com

Alyssa Wilk
Google

Email: alyssar@google.com