QUIC
Internet-Draft
Intended status: Standards Track
Expires: April 26, 2019

J. Iyengar, Ed.
Fastly
M. Thomson, Ed.
Mozilla

October 23, 2018

QUIC: A UDP-Based Multiplexed and Secure Transport
draft-ietf-quic-transport-16

Abstract

This document defines the core of the QUIC transport protocol. This document describes connection establishment, packet format, multiplexing, and reliability. Accompanying documents describe the cryptographic handshake and loss detection.

Note to Readers

Discussion of this draft takes place on the QUIC working group mailing list (quic@ietf.org), which is archived at <https://mailarchive.ietf.org/arch/search/?email_list=quic>.

Working Group information can be found at <https://github.com/quicwg>; source code and issues list for this draft can be found at <https://github.com/quicwg/base-drafts/labels/-transport>.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on April 26, 2019.

Table of Contents

1. Introduction

QUIC is a multiplexed and secure transport protocol that runs on top of UDP. QUIC aims to provide a flexible set of features that allow

it to be a general-purpose secure transport for multiple applications.

- Version negotiation
- Low-latency connection establishment
- Authenticated and encrypted header and payload
- Stream multiplexing
- Stream and connection-level flow control
- Connection migration and resilience to NAT rebinding

QUIC uses UDP as a substrate to avoid requiring changes in legacy client operating systems and middleboxes. QUIC authenticates all of its headers and encrypts most of the data it exchanges, including its signaling. This allows the protocol to evolve without incurring a dependency on upgrades to middleboxes.

1.1. Document Structure

This document describes the core QUIC protocol, and is structured as follows:

- Streams are the basic service abstraction that QUIC provides.
  - Section 2 describes core concepts related to streams,
  - Section 3 provides a reference model for stream states, and
  - Section 4 outlines the operation of flow control.
- Connections are the context in which QUIC endpoints communicate.
  - Section 5 describes core concepts related to connections,
  - Section 6 describes version negotiation,
  - Section 7 details the process for establishing connections,
  - Section 8 specifies critical denial of service mitigation mechanisms,
  - Section 9 describes how endpoints migrate a connection to use a new network paths, and

  * Section 10 lists the options for terminating an open connection.

- Packets and frames are the basic unit used by QUIC to communicate.
  * Section 12 describes concepts related to packets and frames,
  * Section 13 defines models for the transmission, retransmission, and acknowledgement of information, and
  * Section 14 contains a rules for managing the size of packets.
- Details of encoding of QUIC protocol elements is described in:
  * Section 15 (Versions),
  * Section 17 (Packet Headers),
  * Section 18 (Transport Parameters),
  * Section 19 (Frames), and
  * Section 20 (Errors).

Accompanying documents describe QUIC's loss detection and congestion control [QUIC-RECOVERY], and the use of TLS 1.3 for key negotiation [QUIC-TLS].

QUIC version 1 conforms to the protocol invariants in [QUIC-INVARIANTS].

1.2. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

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 unidirectional or bidirectional 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: An opaque identifier that is used to identify a QUIC connection at an endpoint. Each endpoint sets a value that its peer includes in packets.

QUIC packet: The smallest unit of data that can be exchanged by QUIC endpoints.

QUIC is a name, not an acronym.

1.3. Notational Conventions

Packet and frame diagrams use the format described in Section 3.1 of [RFC2360], with the following additional conventions:

[x] Indicates that x is optional

x (A) Indicates that x is A bits long

x (A/B/C) ... Indicates that x is one of A, B, or C bits long

x (i) ... Indicates that x uses the variable-length encoding in Section 16

x (*) ... Indicates that x is variable-length

2. Streams

Streams in QUIC provide a lightweight, ordered byte-stream abstraction.

There are two basic types of stream in QUIC. Unidirectional streams carry data in one direction: from the initiator of the stream to its peer; bidirectional streams allow for data to be sent in both directions. Different stream identifiers are used to distinguish between unidirectional and bidirectional streams, as well as to create a separation between streams that are initiated by the client and server (see Section 2.1).

Either type of stream can be created by either endpoint, can concurrently send data interleaved with other streams, and can be cancelled.

Streams can be created by sending data. Other processes associated with stream management - ending, cancelling, and managing flow control - are all designed to impose minimal overheads. For instance, a single STREAM frame (Section 19.19) can open, carry data for, and close a stream. Streams can also be long-lived and can last the entire duration of a connection.

Stream offsets allow for the octets on a stream to be placed in order. An endpoint MUST be capable of delivering data received on a stream in order. Implementations MAY choose to offer the ability to deliver data out of order. There is no means of ensuring ordering between octets on different streams.

Streams are individually flow controlled, allowing an endpoint to limit memory commitment and to apply back pressure. The creation of streams is also flow controlled, with each peer declaring the maximum stream ID it is willing to accept at a given time.

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

2.1. Stream Identifiers

Streams are identified by an unsigned 62-bit integer, referred to as the Stream ID. Stream IDs are encoded as a variable-length integer (see Section 16). The least significant two bits of the Stream ID are used to identify the type of stream (unidirectional or bidirectional) and the initiator of the stream.

The least significant bit (0x1) of the Stream ID identifies the initiator of the stream. Clients initiate even-numbered streams (those with the least significant bit set to 0); servers initiate odd-numbered streams (with the bit set to 1). Separation of the stream identifiers ensures that client and server are able to open streams without the latency imposed by negotiating for an identifier.

If an endpoint receives a frame for a stream that it expects to initiate (i.e., odd-numbered for the client or even-numbered for the server), but which it has not yet opened, it MUST close the connection with error code STREAM_STATE_ERROR.

The second least significant bit (0x2) of the Stream ID differentiates between unidirectional streams and bidirectional streams. Unidirectional streams always have this bit set to 1 and bidirectional streams have this bit set to 0.

The two type bits from a Stream ID therefore identify streams as summarized in Table 1.

Low Bits	Stream Type
0x0	Client-Initiated, Bidirectional
0x1	Server-Initiated, Bidirectional
0x2	Client-Initiated, Unidirectional
0x3	Server-Initiated, Unidirectional

Table 1: Stream ID Types

The first bidirectional stream opened by the client is stream 0.

A QUIC endpoint MUST NOT reuse a Stream ID. Streams of each type are created in numeric order. Streams that are used out of order result in opening all lower-numbered streams of the same type in the same direction.

2.2. Stream Concurrency

QUIC allows for an arbitrary number of streams to operate concurrently. An endpoint limits the number of concurrently active incoming streams by limiting the maximum stream ID (see Section 4.5).

The maximum stream ID is specific to each endpoint and applies only to the peer that receives the setting. That is, clients specify the maximum stream ID the server can initiate, and servers specify the maximum stream ID the client can initiate. Each endpoint may respond on streams initiated by the other peer, regardless of whether it is permitted to initiate new streams.

Endpoints MUST NOT exceed the limit set by their peer. An endpoint that receives a STREAM frame with an ID greater than the limit it has sent MUST treat this as a stream error of type STREAM_ID_ERROR (Section 11), unless this is a result of a change in the initial limits (see Section 7.3.1).

A receiver cannot renege on an advertisement; that is, once a receiver advertises a stream ID via a MAX_STREAM_ID frame, advertising a smaller maximum ID has no effect. A receiver MUST ignore any MAX_STREAM_ID frame that does not increase the maximum stream ID.

2.3. Sending and Receiving Data

Endpoints uses streams to send and receive data. Endpoints send STREAM frames, which encapsulate data for a stream. STREAM frames carry a flag that can be used to signal the end of a stream.

Streams are an ordered byte-stream abstraction with no other structure that is visible to QUIC. STREAM frame boundaries are not expected to preserved when data is transmitted, when data is retransmitted after packet loss, or when data is delivered 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 octet of this new data. The first octet of data on a stream has an offset of 0. An endpoint is expected to send every stream octet. The largest offset delivered on a stream MUST be less than 2^62.

QUIC makes no specific allowances for partial reliability or delivery of stream data out of order. Endpoints MUST be able to deliver stream data to an application as an ordered byte-stream. Delivering an ordered byte-stream requires that an endpoint buffer any data that is received out of order, up to the advertised flow control limit.

An endpoint could receive the same octets multiple times; octets that have already been received can be discarded. The value for a given octet MUST NOT change if it is sent multiple times; an endpoint MAY treat receipt of a changed octet as a connection error of type PROTOCOL_VIOLATION.

An endpoint MUST NOT send data on any stream without ensuring that it is within the data limits set by its peer. Flow control is described in detail in Section 4.

2.4. Stream Prioritization

Stream multiplexing has a significant effect on application performance if resources allocated to streams are correctly prioritized. Experience with other multiplexed protocols, such as HTTP/2 [HTTP2], shows that effective prioritization strategies have a significant positive impact on performance.

QUIC does not provide frames for exchanging prioritization information. Instead it relies on receiving priority information from the application that uses QUIC. Protocols that use QUIC are able to define any prioritization scheme that suits their application semantics. A protocol might define explicit messages for signaling

priority, such as those defined in HTTP/2; it could define rules that allow an endpoint to determine priority based on context; or it could leave the determination to the application.

A QUIC implementation SHOULD provide ways in which an application can indicate the relative priority of streams. When deciding which streams to dedicate resources to, QUIC SHOULD use the information provided by the application. Failure to account for priority of streams can result in suboptimal performance.

Stream priority is most relevant when deciding which stream data will be transmitted. Often, there will be limits on what can be transmitted as a result of connection flow control or the current congestion controller state.

Giving preference to the transmission of its own management frames ensures that the protocol functions efficiently. That is, prioritizing frames other than STREAM frames ensures that loss recovery, congestion control, and flow control operate effectively.

CRYPTO frames SHOULD be prioritized over other streams prior to the completion of the cryptographic handshake. This includes the retransmission of the second flight of client handshake messages, that is, the TLS Finished and any client authentication messages.

STREAM data in frames determined to be lost SHOULD be retransmitted before sending new data, unless application priorities indicate otherwise. Retransmitting lost stream data can fill in gaps, which allows the peer to consume already received data and free up the flow control window.

3. Stream States: Life of a Stream

This section describes the two types of QUIC stream in terms of the states of their send or receive components. Two state machines are described: one for streams on which an endpoint transmits data (Section 3.1); another for streams from which an endpoint receives data (Section 3.2).

Unidirectional streams use the applicable state machine directly. Bidirectional streams use both state machines. For the most part, the use of these state machines is the same whether the stream is unidirectional or bidirectional. The conditions for opening a stream are slightly more complex for a bidirectional stream because the opening of either send or receive sides causes the stream to open in both directions.

An endpoint can open streams up to its maximum stream limit in any order, however endpoints SHOULD open the send side of streams for each type in order.

Note: These states are largely informative. This document uses stream states to describe rules for when and how different types of frames can be sent and the reactions that are expected when different types of frames are received. Though these state machines are intended to be useful in implementing QUIC, these states aren't intended to constrain implementations. An implementation can define a different state machine as long as its behavior is consistent with an implementation that implements these states.

3.1. Send Stream States

Figure 1 shows the states for the part of a stream that sends data to a peer.

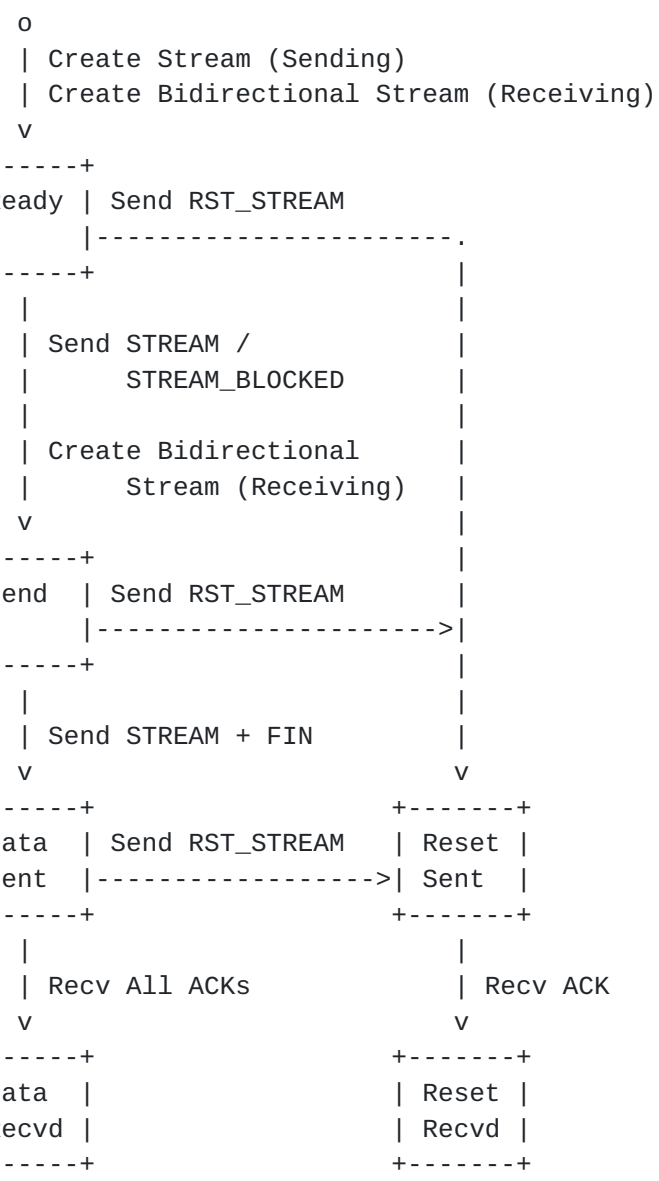


Figure 1: States for Send Streams

The sending part of stream that the endpoint initiates (types 0 and 2 for clients, 1 and 3 for servers) is opened by the application or application protocol. The "Ready" state represents a newly created stream that is able to accept data from the application. Stream data might be buffered in this state in preparation for sending.

Sending the first STREAM or STREAM_BLOCKED frame causes a send stream to enter the "Send" state. An implementation might choose to defer allocating a Stream ID to a send stream until it sends the first frame and enters this state, which can allow for better stream prioritization.

The sending part of a bidirectional stream initiated by a peer (type 0 for a server, type 1 for a client) enters the "Ready" state then immediately transitions to the "Send" state if the receiving part enters the "Recv" state.

In the "Send" state, an endpoint transmits - and retransmits as necessary - data in STREAM frames. The endpoint respects the flow control limits of its peer, accepting MAX_STREAM_DATA frames. An endpoint in the "Send" state generates STREAM_BLOCKED frames if it encounters flow control limits.

After the application indicates that stream data is complete and a STREAM frame containing the FIN bit is sent, the send stream enters the "Data Sent" state. From this state, the endpoint only retransmits stream data as necessary. The endpoint no longer needs to track flow control limits or send STREAM_BLOCKED frames for a send stream in this state. The endpoint can ignore any MAX_STREAM_DATA frames it receives from its peer in this state; MAX_STREAM_DATA frames might be received until the peer receives the final stream offset.

Once all stream data has been successfully acknowledged, the send stream enters the "Data Recvd" state, which is a terminal state.

From any of the "Ready", "Send", or "Data Sent" states, an application can signal that it wishes to abandon transmission of stream data. Similarly, the endpoint might receive a STOP_SENDING frame from its peer. In either case, the endpoint sends a RST_STREAM frame, which causes the stream to enter the "Reset Sent" state.

An endpoint MAY send a RST_STREAM as the first frame on a send stream; this causes the send stream to open and then immediately transition to the "Reset Sent" state.

Once a packet containing a RST_STREAM has been acknowledged, the send stream enters the "Reset Recvd" state, which is a terminal state.

3.2. Receive Stream States

Figure 2 shows the states for the part of a stream that receives data from a peer. The states for a receive stream mirror only some of the states of the send stream at the peer. A receive stream doesn't track states on the send stream that cannot be observed, such as the "Ready" state; instead, receive streams track the delivery of data to the application or application protocol some of which cannot be observed by the sender.

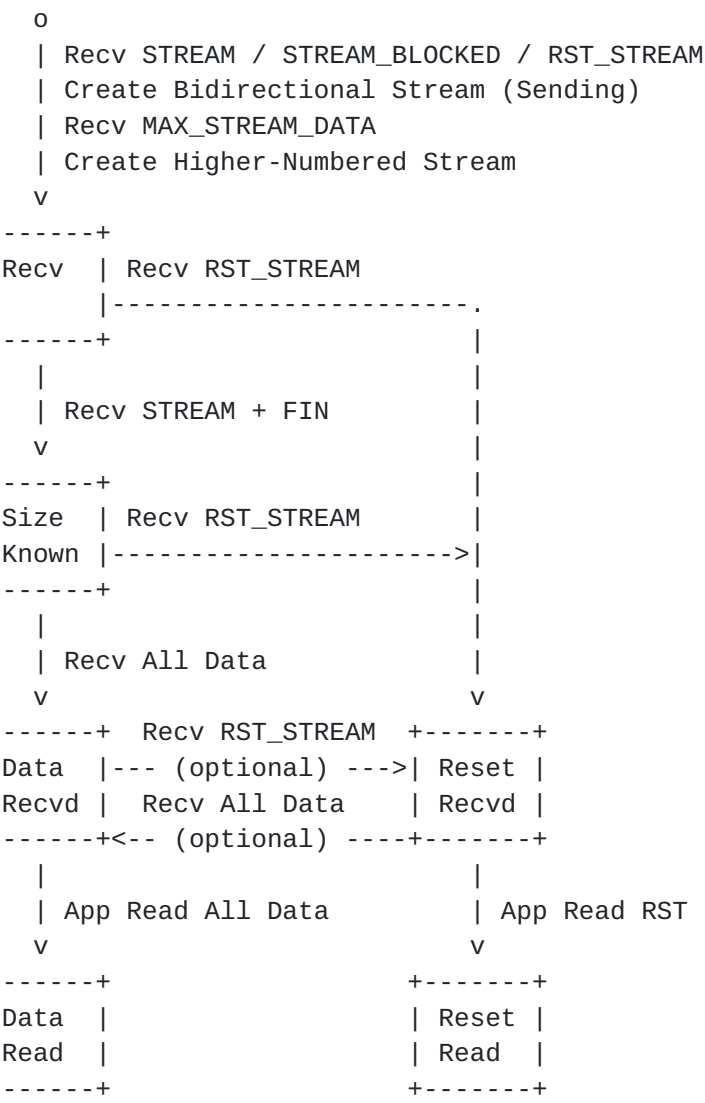


Figure 2: States for Receive Streams

The receiving part of a stream initiated by a peer (types 1 and 3 for a client, or 0 and 2 for a server) are created when the first STREAM, STREAM_BLOCKED, RST_STREAM, or MAX_STREAM_DATA (bidirectional only, see below) is received for that stream. The initial state for a receive stream is "Recv". Receiving a RST_STREAM frame causes the receive stream to immediately transition to the "Reset Recvd".

The receive stream enters the "Recv" state when the sending part of a bidirectional stream initiated by the endpoint (type 0 for a client, type 1 for a server) enters the "Ready" state.

A bidirectional stream also opens when a MAX_STREAM_DATA frame is received. Receiving a MAX_STREAM_DATA frame implies that the remote peer has opened the stream and is providing flow control credit. A

MAX_STREAM_DATA frame might arrive before a STREAM or STREAM_BLOCKED frame if packets are lost or reordered.

Before creating a stream, all lower-numbered streams of the same type MUST be created. That means that receipt of a frame that would open a stream causes all lower-numbered streams of the same type to be opened in numeric order. This ensures that the creation order for streams is consistent on both endpoints.

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

When a STREAM frame with a FIN bit is received, the final offset (see Section 4.3) is known. The receive stream enters the "Size Known" state. In this state, the endpoint no longer needs to send MAX_STREAM_DATA frames, it only receives any retransmissions of stream data.

Once all data for the stream has been received, the receive stream enters the "Data Recvd" state. This might happen as a result of receiving the same STREAM frame that causes the transition to "Size Known". In this state, the endpoint has all stream data. Any STREAM or STREAM_BLOCKED frames it receives for the stream can be discarded.

The "Data Recvd" state persists until stream data has been delivered to the application or application protocol. Once stream data has been delivered, the stream enters the "Data Read" state, which is a terminal state.

Receiving a RST_STREAM frame in the "Recv" or "Size Known" states causes the stream to enter the "Reset Recvd" state. This might cause the delivery of stream data to the application to be interrupted.

It is possible that all stream data is received when a RST_STREAM is received (that is, from the "Data Recvd" state). Similarly, it is possible for remaining stream data to arrive after receiving a RST_STREAM frame (the "Reset Recvd" state). An implementation is able to manage this situation as they choose. Sending RST_STREAM means that an endpoint cannot guarantee delivery of stream data; however there is no requirement that stream data not be delivered if a RST_STREAM is received. An implementation MAY interrupt delivery of stream data, discard any data that was not consumed, and signal the existence of the RST_STREAM immediately. Alternatively, the RST_STREAM signal might be suppressed or withheld if stream data is

completely received. In the latter case, the receive stream effectively transitions to "Data Recvd" from "Reset Recvd".

Once the application has been delivered the signal indicating that the receive stream was reset, the receive stream transitions to the "Reset Read" state, which is a terminal state.

3.3. Permitted Frame Types

The sender of a stream sends just three frame types that affect the state of a stream at either sender or receiver: STREAM (Section 19.19), STREAM_BLOCKED (Section 19.10), and RST_STREAM (Section 19.2).

A sender MUST NOT send any of these frames from a terminal state ("Data Recvd" or "Reset Recvd"). A sender MUST NOT send STREAM or STREAM_BLOCKED after sending a RST_STREAM; that is, in the "Reset Sent" state in addition to the terminal states. A receiver could receive any of these frames in any state, but only due to the possibility of delayed delivery of packets carrying them.

The receiver of a stream sends MAX_STREAM_DATA (Section 19.6) and STOP_SENDING frames (Section 19.14).

The receiver only sends MAX_STREAM_DATA in the "Recv" state. A receiver can send STOP_SENDING in any state where it has not received a RST_STREAM frame; that is states other than "Reset Recvd" or "Reset Read". However there is little value in sending a STOP_SENDING frame after all stream data has been received in the "Data Recvd" state. A sender could receive these frames in any state as a result of delayed delivery of packets.

3.4. Bidirectional Stream States

A bidirectional stream is composed of a send stream and a receive stream. Implementations may represent states of the bidirectional stream as composites of send and receive stream states. The simplest model presents the stream as "open" when either send or receive stream is in a non-terminal state and "closed" when both send and receive streams are in a terminal state.

Table 2 shows a more complex mapping of bidirectional stream states that loosely correspond to the stream states in HTTP/2 [HTTP2]. This shows that multiple states on send or receive streams are mapped to the same composite state. Note that this is just one possibility for such a mapping; this mapping requires that data is acknowledged before the transition to a "closed" or "half-closed" state.

Send Stream	Receive Stream	Composite State
No Stream/Ready	No Stream/Recv *1	idle
Ready/Send/Data Sent	Recv/Size Known	open
Ready/Send/Data Sent	Data Recvd/Data Read	half-closed (remote)
Ready/Send/Data Sent	Reset Recvd/Reset Read	half-closed (remote)
Data Recvd	Recv/Size Known	half-closed (local)
Reset Sent/Reset Recvd	Recv/Size Known	half-closed (local)
Data Recvd	Recv/Size Known	half-closed (local)
Reset Sent/Reset Recvd	Data Recvd/Data Read	closed
Reset Sent/Reset Recvd	Reset Recvd/Reset Read	closed
Data Recvd	Data Recvd/Data Read	closed
Data Recvd	Reset Recvd/Reset Read	closed

Table 2: Possible Mapping of Stream States to HTTP/2

Note (*1): A stream is considered "idle" if it has not yet been created, or if the receive stream is in the "Recv" state without yet having received any frames.

3.5. Solicited State Transitions

If an endpoint is no longer interested in the data it is receiving on a stream, it MAY send a STOP_SENDING frame identifying that stream to prompt closure of the stream in the opposite direction. This typically indicates that the receiving application is no longer reading data it receives from the stream, but is not a guarantee that incoming data will be ignored.

STREAM frames received after sending STOP_SENDING are still counted toward the connection and stream flow-control windows, even though these frames will be discarded upon receipt. This avoids potential ambiguity about which STREAM frames count toward flow control.

A STOP_SENDING frame requests that the receiving endpoint send a RST_STREAM frame. An endpoint that receives a STOP_SENDING frame MUST send a RST_STREAM frame for that stream, and can use an error code of STOPPING. If the STOP_SENDING frame is received on a send stream that is already in the "Data Sent" state, a RST_STREAM frame MAY still be sent in order to cancel retransmission of previously-sent STREAM frames.

STOP_SENDING SHOULD only be sent for a receive stream that has not been reset. STOP_SENDING is most useful for streams in the "Recv" or "Size Known" states.

An endpoint is expected to send another STOP_SENDING frame if a packet containing a previous STOP_SENDING is lost. However, once either all stream data or a RST_STREAM frame has been received for the stream - that is, the stream is in any state other than "Recv" or "Size Known" - sending a STOP_SENDING frame is unnecessary.

4. Flow Control

It is necessary to limit the amount of data that a sender may have outstanding at any time, so as to prevent a fast sender from overwhelming a slow receiver, or to prevent a malicious sender from consuming significant resources at a receiver. To this end, QUIC employs a credit-based flow-control scheme similar to that in HTTP/2 [HTTP2]. 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:

- Stream flow control, which prevents a single stream from consuming the entire receive buffer for a connection.

- Connection flow control, which prevents senders from exceeding a receiver's buffer capacity for the connection, and

A data receiver sets initial credits for all streams by sending transport parameters during the handshake (Section 7.3).

A data receiver sends MAX_STREAM_DATA or MAX_DATA frames to the sender to advertise additional credit. MAX_STREAM_DATA frames send the maximum absolute byte offset of a stream, while MAX_DATA frames send the maximum of the sum of the absolute byte offsets of all streams.

A receiver advertises credit for a stream by sending a MAX_STREAM_DATA frame with the Stream ID set appropriately. A receiver could use the current offset of data consumed to determine the flow control offset to be advertised. A receiver MAY send MAX_STREAM_DATA frames in multiple packets in order to make sure that the sender receives an update before running out of flow control credit, even if one of the packets is lost.

Connection flow control is a limit to the total bytes of stream data sent in STREAM frames on all streams. A receiver advertises credit for a connection by sending a MAX_DATA frame. A receiver maintains a cumulative sum of bytes received on all contributing streams, which are used to check for flow control violations. A receiver might use a sum of bytes consumed on all streams to determine the maximum data limit to be advertised.

A receiver MAY advertise a larger offset at any point by sending MAX_STREAM_DATA or MAX_DATA frames. A receiver cannot renege on an advertisement; that is, once a receiver advertises an offset, advertising a smaller offset has no effect. A sender MUST therefore ignore any MAX_STREAM_DATA or MAX_DATA frames that do not increase flow control limits.

A receiver MUST close the connection with a FLOW_CONTROL_ERROR error (Section 11) if the peer violates the advertised connection or stream data limits.

A sender SHOULD send STREAM_BLOCKED or BLOCKED frames to indicate it has data to write but is blocked by flow control limits. These frames are expected to be sent infrequently in common cases, but they are considered useful for debugging and monitoring purposes.

A similar method is used to control the number of open streams (see Section 4.5 for details).

4.1. Handling of Stream Cancellation

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 MAX_STREAM_DATA or MAX_DATA frame which will never come.

On receipt of a 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 the number of bytes that were sent on the stream to make the same adjustment in its connection flow controller.

To ensure that endpoints maintain a consistent connection-level flow control state, the RST_STREAM frame (Section 19.2) includes the largest offset of data sent on the stream. 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.

RST_STREAM terminates one direction of a stream abruptly. Whether any action or response can or should be taken on the data already received is application specific.

For a bidirectional stream, RST_STREAM has no effect on data flow in the opposite direction. The RST_STREAM sender can send a STOP_SENDING frame to encourage prompt termination. Both endpoints MUST maintain state for the stream in the unterminated direction until that direction enters a terminal state, or either side sends CONNECTION_CLOSE or APPLICATION_CLOSE.

4.2. Data Limit Increments

This document leaves when and how many bytes to advertise in a MAX_DATA or MAX_STREAM_DATA to implementations, but offers a few considerations. These frames contribute to connection overhead. Therefore frequently sending frames with small changes is undesirable. At the same time, larger increments to limits are necessary to avoid blocking if updates are less frequent, requiring larger resource commitments at the receiver. Thus there is a trade-off between resource commitment and overhead when determining how large a limit is advertised.

A receiver MAY use an autotuning mechanism to tune the frequency and amount that it increases data limits based on a round-trip time estimate and the rate at which the receiving application consumes data, similar to common TCP implementations.

If a sender runs out of flow control credit, it will be unable to send new data. That is, the sender is blocked. A blocked sender SHOULD send a STREAM_BLOCKED or BLOCKED frame. A receiver uses these frames for debugging purposes. A receiver MUST NOT wait for a STREAM_BLOCKED or BLOCKED frame before sending MAX_STREAM_DATA or MAX_DATA, since doing so will mean that a sender will be blocked for an entire round trip and the peer may never send a STREAM_BLOCKED or BLOCKED frame.

It is generally considered best to not let the sender go into quiescence if avoidable. To avoid blocking a sender, and to reasonably account for the possibility of loss, a receiver should send a MAX_DATA or MAX_STREAM_DATA frame at least two round trips before it expects the sender to get blocked.

A sender sends a single BLOCKED or STREAM_BLOCKED frame only once when it reaches a data limit. A sender SHOULD NOT send multiple BLOCKED or STREAM_BLOCKED frames for the same data limit, unless the original frame is determined to be lost. Another BLOCKED or STREAM_BLOCKED frame can be sent after the data limit is increased.

4.3. Stream Final Offset

The final offset is the count of the number of octets that are transmitted on a stream. For a stream that is reset, the final offset is carried explicitly in a RST_STREAM frame. Otherwise, the final offset is the offset of the end of the data carried in a STREAM frame marked with a FIN flag, or 0 in the case of incoming unidirectional streams.

An endpoint will know the final offset for a stream when the receive stream enters the "Size Known" or "Reset Recvd" state.

An endpoint MUST NOT send data on a stream at or beyond the final offset.

Once a final offset for a stream is known, it cannot change. If a RST_STREAM or STREAM frame causes the final offset to change for a stream, an endpoint SHOULD respond with a FINAL_OFFSET_ERROR error (see Section 11). A receiver SHOULD treat receipt of data at or beyond the final offset as a FINAL_OFFSET_ERROR error, even after a stream is closed. Generating these errors is not mandatory, but only because requiring that an endpoint generate these errors also means that the endpoint needs to maintain the final offset state for closed streams, which could mean a significant state commitment.

4.4. Flow Control for Cryptographic Handshake

Data sent in CRYPTO frames is not flow controlled in the same way as STREAM frames. QUIC relies on the cryptographic protocol implementation to avoid excessive buffering of data, see [QUIC-TLS]. The implementation SHOULD provide an interface to QUIC to tell it about its buffering limits so that there is not excessive buffering at multiple layers.

4.5. Stream Limit Increment

An endpoint limits the number of concurrently active incoming streams by limiting the maximum stream ID. An initial value is set in the transport parameters (see Section 18.1) and is subsequently increased by MAX_STREAM_ID frames (see Section 19.7).

As with stream and connection flow control, this document leaves when and how many streams to make available to a peer via MAX_STREAM_ID to implementations, but offers a few considerations. MAX_STREAM_ID frames constitute minimal overhead, while withholding MAX_STREAM_ID frames can prevent the peer from using the available parallelism.

The STREAM_ID_BLOCKED frame (Section 19.11) can be used to signal a shortage of available streams. Implementations will likely want to increase the maximum stream ID as peer-initiated streams close.

5. Connections

A QUIC connection is a single conversation between two QUIC endpoints. QUIC's connection establishment combines version negotiation with the cryptographic and transport handshakes to reduce connection establishment latency, as described in Section 7. Once established, a connection may migrate to a different IP or port at either endpoint as described in Section 9. Finally, a connection may be terminated by either endpoint, as described in Section 10.

5.1. Connection ID

Each connection possesses a set of connection identifiers, or connection IDs, each of which can be identify the connection. Connection IDs are independently selected by endpoints; each endpoint selects the connection IDs that its peer uses.

The primary function of a connection ID is to ensure that changes in addressing at lower protocol layers (UDP, IP, and below) don't cause packets for a QUIC connection to be delivered to the wrong endpoint. Each endpoint selects connection IDs using an implementation-specific (and perhaps deployment-specific) method which will allow packets

with that connection ID to be routed back to the endpoint and identified by the endpoint upon receipt.

Connection IDs MUST NOT contain any information that can be used to correlate them with other connection IDs for the same connection. As a trivial example, this means the same connection ID MUST NOT be issued more than once on the same connection.

Packets with long headers include Source Connection ID and Destination Connection ID fields. These fields are used to set the connection IDs for new connections, see Section 7.2 for details.

Packets with short headers (Section 17.3) only include the Destination Connection ID and omit the explicit length. The length of the Destination Connection ID field is expected to be known to endpoints. Endpoints using a load balancer that routes based on connection ID could agree with the load balancer on a fixed length for connection IDs, or agree on an encoding scheme. A fixed portion could encode an explicit length, which allows the entire connection ID to vary in length and still be used by the load balancer.

A Version Negotiation (Section 17.4) packet echoes the connection IDs selected by the client, both to ensure correct routing toward the client and to allow the client to validate that the packet is in response to an Initial packet.

A zero-length connection ID MAY be used when the connection ID is not needed for routing and the address/port tuple of packets is sufficient to identify a connection. An endpoint whose peer has selected a zero-length connection ID MUST continue to use a zero-length connection ID for the lifetime of the connection and MUST NOT send packets from any other local address.

When an endpoint has requested a non-zero-length connection ID, it needs to ensure that the peer has a supply of connection IDs from which to choose for packets sent to the endpoint. These connection IDs are supplied by the endpoint using the NEW_CONNECTION_ID frame (Section 19.12).

5.1.1. Issuing Connection IDs

Each Connection ID has an associated sequence number to assist in deduplicating messages. The initial connection ID issued by an endpoint is sent in the Source Connection ID field of the long packet header (Section 17.2) during the handshake. The sequence number of the initial connection ID is 0. If the preferred_address transport parameter is sent, the sequence number of the supplied connection ID is 1.

Additional connection IDs are communicated to the peer using NEW_CONNECTION_ID frames (Section 19.12). The sequence number on each newly-issued connection ID MUST increase by 1. The connection ID randomly selected by the client in the Initial packet and any connection ID provided by a Reset packet are not assigned sequence numbers unless a server opts to retain them as its initial connection ID.

When an endpoint issues a connection ID, it MUST accept packets that carry this connection ID for the duration of the connection or until its peer invalidates the connection ID via a RETIRE_CONNECTION_ID frame (Section 19.13).

An endpoint SHOULD ensure that its peer has a sufficient number of available and unused connection IDs. While each endpoint independently chooses how many connection IDs to issue, endpoints SHOULD provide and maintain at least eight connection IDs. The endpoint can do this by always supplying a new connection ID when a connection ID is retired by its peer or when the endpoint receives a packet with a previously unused connection ID. Endpoints that initiate migration and require non-zero-length connection IDs SHOULD provide their peers with new connection IDs before migration, or risk the peer closing the connection.

5.1.2. Consuming and Retiring Connection IDs

An endpoint can change the connection ID it uses for a peer to another available one at any time during the connection. An endpoint consumes connection IDs in response to a migrating peer, see Section 9.5 for more.

An endpoint maintains a set of connection IDs received from its peer, any of which it can use when sending packets. When the endpoint wishes to remove a connection ID from use, it sends a RETIRE_CONNECTION_ID frame to its peer, indicating that the peer might bring a new connection ID into circulation using the NEW_CONNECTION_ID frame.

An endpoint that retires a connection ID can retain knowledge of that connection ID for a period of time after sending the RETIRE_CONNECTION_ID frame, or until that frame is acknowledged.

As discussed in Section 9.5, each connection ID MUST be used on packets sent from only one local address. An endpoint that migrates away from a local address SHOULD retire all connection IDs used on that address once it no longer plans to use that address.

5.2. Matching Packets to Connections

Incoming packets are classified on receipt. Packets can either be associated with an existing connection, or - for servers - potentially create a new connection.

Hosts try to associate a packet with an existing connection. If the packet has a Destination Connection ID corresponding to an existing connection, QUIC processes that packet accordingly. Note that more than one connection ID can be associated with a connection; see Section 5.1.

If the Destination Connection ID is zero length and the packet matches the address/port tuple of a connection where the host did not require connection IDs, QUIC processes the packet as part of that connection. Endpoints MUST drop packets with zero-length Destination Connection ID fields if they do not correspond to a single connection.

Endpoints SHOULD send a Stateless Reset (Section 10.4) for any packets that cannot be attributed to an existing connection.

Packets that are matched to an existing connection, but for which the endpoint cannot remove packet protection, are discarded.

5.2.1. Client Packet Handling

Valid packets sent to clients always include a Destination Connection ID that matches a value the client selects. Clients that choose to receive zero-length connection IDs can use the address/port tuple to identify a connection. Packets that don't match an existing connection are discarded.

Due to packet reordering or loss, clients might receive packets for a connection that are encrypted with a key it has not yet computed. Clients MAY drop these packets, or MAY buffer them in anticipation of later packets that allow it to compute the key.

If a client receives a packet that has an unsupported version, it MUST discard that packet.

5.2.2. Server Packet Handling

If a server receives a packet that has an unsupported version, but the packet is sufficiently large to initiate a new connection for any version supported by the server, it SHOULD send a Version Negotiation packet as described in Section 6.1. Servers MAY rate control these packets to avoid storms of Version Negotiation packets.

The first packet for an unsupported version can use different semantics and encodings for any version-specific field. In particular, different packet protection keys might be used for different versions. Servers that do not support a particular version are unlikely to be able to decrypt the payload of the packet. Servers SHOULD NOT attempt to decode or decrypt a packet from an unknown version, but instead send a Version Negotiation packet, provided that the packet is sufficiently long.

Servers MUST drop other packets that contain unsupported versions.

Packets with a supported version, or no version field, are matched to a connection using the connection ID or - for packets with zero-length connection IDs - the address tuple. If the packet doesn't match an existing connection, the server continues below.

If the packet is an Initial packet fully conforming with the specification, the server proceeds with the handshake (Section 7). This commits the server to the version that the client selected.

If a server isn't currently accepting any new connections, it SHOULD send an Initial packet containing a CONNECTION_CLOSE frame with error code SERVER_BUSY.

If the packet is a 0-RTT packet, the server MAY buffer a limited number of these packets in anticipation of a late-arriving Initial Packet. Clients are forbidden from sending Handshake packets prior to receiving a server response, so servers SHOULD ignore any such packets.

Servers MUST drop incoming packets under all other circumstances.

5.3. Life of a QUIC Connection

TBD.

6. Version Negotiation

Version negotiation ensures that client and server agree to a QUIC version that is mutually supported. A server sends a Version Negotiation packet in response to each packet that might initiate a new connection, see Section 5.2 for details.

The first few messages of an exchange between a client attempting to create a new connection with server is shown in Figure 3. After version negotiation completes, connection establishment can proceed, for example as shown in Section 7.1.

Figure 3: Example Version Negotiation Exchange

The size of the first packet sent by a client will determine whether a server sends a Version Negotiation packet. Clients that support multiple QUIC versions SHOULD pad the first packet they send to the largest of the minimum packet sizes across all versions they support. This ensures that the server responds if there is a mutually supported version.

6.1. Sending Version Negotiation Packets

If the version selected by the client is not acceptable to the server, the server responds with a Version Negotiation packet (see Section 17.4). This includes a list of versions that the server will accept.

This system allows a server to process packets with unsupported versions without retaining state. Though either the Initial packet or the Version Negotiation packet that is sent in response could be lost, the client will send new packets until it successfully receives a response or it abandons the connection attempt.

A server MAY limit the number of Version Negotiation packets it sends. For instance, a server that is able to recognize packets as 0-RTT might choose not to send Version Negotiation packets in response to 0-RTT packets with the expectation that it will eventually receive an Initial packet.

6.2. Handling Version Negotiation Packets

When the client receives a Version Negotiation packet, it first checks that the Destination and Source Connection ID fields match the Source and Destination Connection ID fields in a packet that the client sent. If this check fails, the packet MUST be discarded.

Once the Version Negotiation packet is determined to be valid, the client then selects an acceptable protocol version from the list provided by the server. The client then attempts to create a

connection using that version. Though the content of the Initial packet the client sends might not change in response to version negotiation, a client MUST increase the packet number it uses on every packet it sends. Packets MUST continue to use long headers (Section 17.2) and MUST include the new negotiated protocol version.

The client MUST use the long header format and include its selected version on all packets until it has 1-RTT keys and it has received a packet from the server which is not a Version Negotiation packet.

A client MUST NOT change the version it uses unless it is in response to a Version Negotiation packet from the server. Once a client receives a packet from the server which is not a Version Negotiation packet, it MUST discard other Version Negotiation packets on the same connection. Similarly, a client MUST ignore a Version Negotiation packet if it has already received and acted on a Version Negotiation packet.

A client MUST ignore a Version Negotiation packet that lists the client's chosen version.

A client MAY attempt 0-RTT after receiving a Version Negotiation packet. A client that sends additional 0-RTT packets MUST NOT reset the packet number to 0 as a result, see Section 17.5.2.

Version negotiation packets have no cryptographic protection. The result of the negotiation MUST be revalidated as part of the cryptographic handshake (see Section 7.3.3).

6.3. Using Reserved Versions

For a server to use a new version in the future, clients must correctly handle unsupported versions. To help ensure this, a server SHOULD include a reserved version (see Section 15) while generating a Version Negotiation packet.

The design of version negotiation permits a server to avoid maintaining state for packets that it rejects in this fashion. The validation of version negotiation (see Section 7.3.3) only validates the result of version negotiation, which is the same no matter which reserved version was sent. A server MAY therefore send different reserved version numbers in the Version Negotiation Packet and in its transport parameters.

A client MAY send a packet using a reserved version number. This can be used to solicit a list of supported versions from a server.

7. Cryptographic and Transport Handshake

QUIC relies on a combined cryptographic and transport handshake to minimize connection establishment latency. QUIC uses the CRYPTO frame Section 19.20 to transmit the cryptographic handshake. Version 0x00000001 of QUIC uses TLS 1.3 as described in [QUIC-TLS]; a different QUIC version number could indicate that a different cryptographic handshake protocol is in use.

QUIC provides reliable, ordered delivery of the cryptographic handshake data. QUIC packet protection ensures confidentiality and integrity protection that meets the requirements of the cryptographic handshake protocol:

- authenticated key exchange, where
  - a server is always authenticated,
  - a client is optionally authenticated,
  - every connection produces distinct and unrelated keys,
  - keying material is usable for packet protection for both 0-RTT and 1-RTT packets, and
  - 1-RTT keys have forward secrecy
- authenticated values for the transport parameters of the peer (see Section 7.3)
- authenticated confirmation of version negotiation (see Section 7.3.3)
- authenticated negotiation of an application protocol (TLS uses ALPN [RFC7301] for this purpose)

The first CRYPTO frame from a client MUST be sent in a single packet. Any second attempt that is triggered by address validation (see Section 8.1) MUST also be sent within a single packet. This avoids having to reassemble a message from multiple packets.

The first client packet of the cryptographic handshake protocol MUST fit within a 1232 octet QUIC packet payload. This includes overheads that reduce the space available to the cryptographic handshake protocol.

The CRYPTO frame can be sent in different packet number spaces. The sequence numbers used by CRYPTO frames to ensure ordered delivery of

cryptographic handshake data start from zero in each packet number space.

7.1. Example Handshake Flows

Details of how TLS is integrated with QUIC are provided in [QUIC-TLS], but some examples are provided here. An extension of this exchange to support client address validation is shown in Section 8.1.1.

Once any version negotiation and address validation exchanges are complete, the cryptographic handshake is used to agree on cryptographic keys. The cryptographic handshake is carried in Initial (Section 17.5) and Handshake (Section 17.6) packets.

Figure 4 provides an overview of the 1-RTT handshake. Each line shows a QUIC packet with the packet type and packet number shown first, followed by the frames that are typically contained in those packets. So, for instance the first packet is of type Initial, with packet number 0, and contains a CRYPTO frame carrying the ClientHello.

Note that multiple QUIC packets - even of different encryption levels - may be coalesced into a single UDP datagram (see Section 12.2), and so this handshake may consist of as few as 4 UDP datagrams, or any number more. For instance, the server's first flight contains packets from the Initial encryption level (obfuscation), the Handshake level, and "0.5-RTT data" from the server at the 1-RTT encryption level.

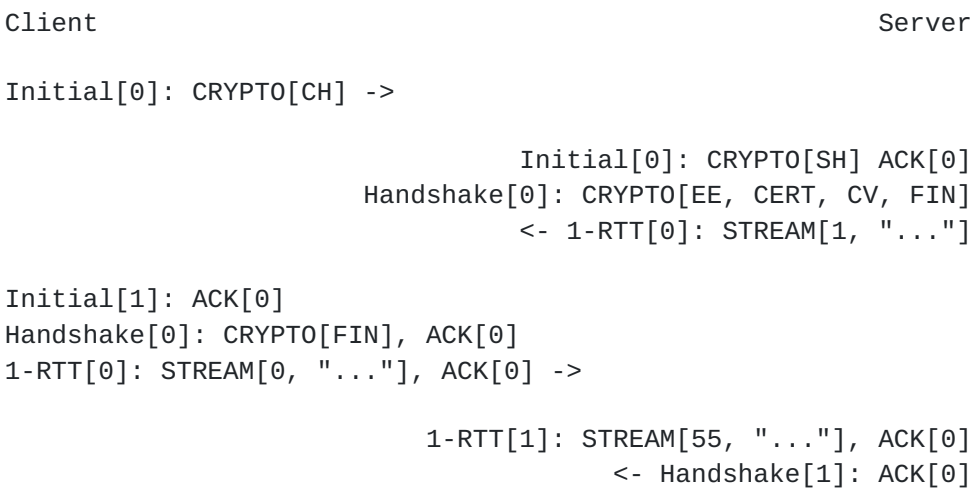


Figure 4: Example 1-RTT Handshake

Figure 5 shows an example of a connection with a 0-RTT handshake and a single packet of 0-RTT data. Note that as described in Section 12.3, the server acknowledges 0-RTT data at the 1-RTT encryption level, and the client sends 1-RTT packets in the same packet number space.

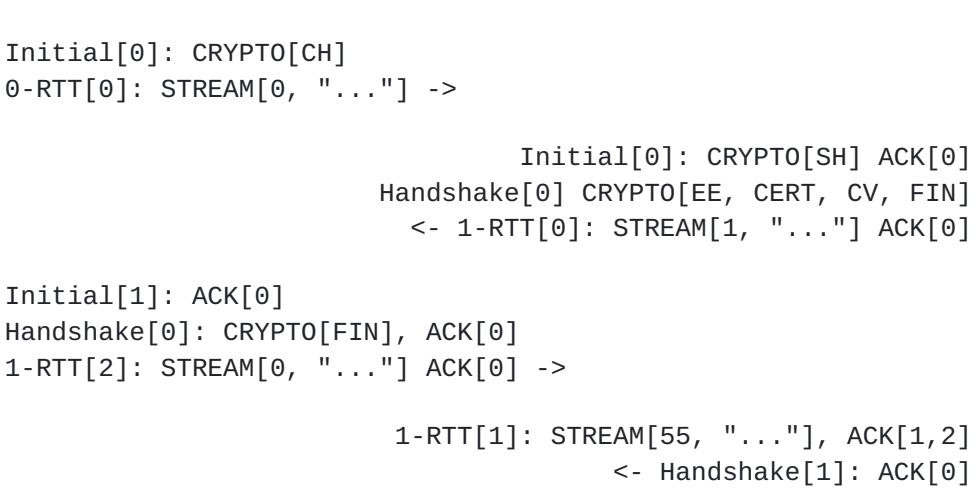


Figure 5: Example 0-RTT Handshake

7.2. Negotiating Connection IDs

A connection ID is used to ensure consistent routing of packets, as described in Section 5.1. The long header contains two connection IDs: the Destination Connection ID is chosen by the recipient of the packet and is used to provide consistent routing; the Source Connection ID is used to set the Destination Connection ID used by the peer.

During the handshake, packets with the long header (Section 17.2) are used to establish the connection ID that each endpoint uses. Each endpoint uses the Source Connection ID field to specify the connection ID that is used in the Destination Connection ID field of packets being sent to them. Upon receiving a packet, each endpoint sets the Destination Connection ID it sends to match the value of the Source Connection ID that they receive.

When an Initial packet is sent by a client which has not previously received a Retry packet from the server, it populates the Destination Connection ID field with an unpredictable value. This MUST be at least 8 octets in length. Until a packet is received from the server, the client MUST use the same value unless it abandons the connection attempt and starts a new one. The initial Destination

Connection ID is used to determine packet protection keys for Initial packets.

The client populates the Source Connection ID field with a value of its choosing and sets the SCIL field to match.

The Destination Connection ID field in the server's Initial packet contains a connection ID that is chosen by the recipient of the packet (i.e., the client); the Source Connection ID includes the connection ID that the sender of the packet wishes to use (see Section 5.1). The server MUST use consistent Source Connection IDs during the handshake.

On first receiving an Initial or Retry packet from the server, the client uses the Source Connection ID supplied by the server as the Destination Connection ID for subsequent packets. That means that a client might change the Destination Connection ID twice during connection establishment. Once a client has received an Initial packet from the server, it MUST discard any packet it receives with a different Source Connection ID.

A client MUST only change the value it sends in the Destination Connection ID in response to the first packet of each type it receives from the server (Retry or Initial); a server MUST set its value based on the Initial packet. Any additional changes are not permitted; if subsequent packets of those types include a different Source Connection ID, they MUST be discarded. This avoids problems that might arise from stateless processing of multiple Initial packets producing different connection IDs.

The connection ID can change over the lifetime of a connection, especially in response to connection migration (Section 9), see Section 5.1.1 for details.

7.3. Transport Parameters

During connection establishment, both endpoints make authenticated declarations of their transport parameters. These declarations are made unilaterally by each endpoint. Endpoints are required to comply with the restrictions implied by these parameters; the description of each parameter includes rules for its handling.

The encoding of the transport parameters is detailed in Section 18.

QUIC includes the encoded transport parameters in the cryptographic handshake. Once the handshake completes, the transport parameters declared by the peer are available. Each endpoint validates the value provided by its peer. In particular, version negotiation MUST

be validated (see Section 7.3.3) before the connection establishment is considered properly complete.

Definitions for each of the defined transport parameters are included in Section 18.1. Any given parameter MUST appear at most once in a given transport parameters extension. An endpoint MUST treat receipt of duplicate transport parameters as a connection error of type TRANSPORT_PARAMETER_ERROR.

A server MUST include the original_connection_id transport parameter (Section 18.1) if it sent a Retry packet.

7.3.1. Values of Transport Parameters for 0-RTT

A client that attempts to send 0-RTT data MUST remember the transport parameters used by the server. The transport parameters that the server advertises during connection establishment apply to all connections that are resumed using the keying material established during that handshake. Remembered transport parameters apply to the new connection until the handshake completes and new transport parameters from the server can be provided.

A server can remember the transport parameters that it advertised, or store an integrity-protected copy of the values in the ticket and recover the information when accepting 0-RTT data. A server uses the transport parameters in determining whether to accept 0-RTT data.

A server MAY accept 0-RTT and subsequently provide different values for transport parameters for use in the new connection. If 0-RTT data is accepted by the server, the server MUST NOT reduce any limits or alter any values that might be violated by the client with its 0-RTT data. In particular, a server that accepts 0-RTT data MUST NOT set values for initial_max_data, initial_max_stream_data_bidi_local, initial_max_stream_data_bidi_remote, initial_max_stream_data_uni, initial_max_bidi_streams, or initial_max_uni_streams (Section 18.1) that are smaller than the remembered value of those parameters.

Omitting or setting a zero value for certain transport parameters can result in 0-RTT data being enabled, but not usable. The applicable subset of transport parameters that permit sending of application data SHOULD be set to non-zero values for 0-RTT. This includes initial_max_data and either initial_max_bidi_streams and initial_max_stream_data_bidi_remote, or initial_max_uni_streams and initial_max_stream_data_uni.

The value of the server's previous preferred_address MUST NOT be used when establishing a new connection; rather, the client should wait to observe the server's new preferred_address value in the handshake.

A server MUST reject 0-RTT data or even abort a handshake if the implied values for transport parameters cannot be supported.

7.3.2. New Transport Parameters

New transport parameters can be used to negotiate new protocol behavior. An endpoint MUST ignore transport parameters that it does not support. Absence of a transport parameter therefore disables any optional protocol feature that is negotiated using the parameter.

New transport parameters can be registered according to the rules in Section 22.1.

7.3.3. Version Negotiation Validation

Though the cryptographic handshake has integrity protection, two forms of QUIC version downgrade are possible. In the first, an attacker replaces the QUIC version in the Initial packet. In the second, a fake Version Negotiation packet is sent by an attacker. To protect against these attacks, the transport parameters include three fields that encode version information. These parameters are used to retroactively authenticate the choice of version (see Section 6).

The cryptographic handshake provides integrity protection for the negotiated version as part of the transport parameters (see Section 18.1). As a result, attacks on version negotiation by an attacker can be detected.

The client includes the initial_version field in its transport parameters. The initial_version is the version that the client initially attempted to use. If the server did not send a Version Negotiation packet Section 17.4, this will be identical to the negotiated_version field in the server transport parameters.

A server that processes all packets in a stateful fashion can remember how version negotiation was performed and validate the initial_version value.

A server that does not maintain state for every packet it receives (i.e., a stateless server) uses a different process. If the initial_version matches the version of QUIC that is in use, a stateless server can accept the value.

If the initial_version is different from the version of QUIC that is in use, a stateless server MUST check that it would have sent a Version Negotiation packet if it had received a packet with the indicated initial_version. If a server would have accepted the version included in the initial_version and the value differs from

the QUIC version that is in use, the server MUST terminate the connection with a VERSION_NEGOTIATION_ERROR error.

The server includes both the version of QUIC that is in use and a list of the QUIC versions that the server supports (see Section 18.1).

The negotiated_version field is the version that is in use. This MUST be set by the server to the value that is on the Initial packet that it accepts (not an Initial packet that triggers a Retry or Version Negotiation packet). A client that receives a negotiated_version that does not match the version of QUIC that is in use MUST terminate the connection with a VERSION_NEGOTIATION_ERROR error code.

The server includes a list of versions that it would send in any version negotiation packet (Section 17.4) in the supported_versions field. The server populates this field even if it did not send a version negotiation packet.

The client validates that the negotiated_version is included in the supported_versions list and - if version negotiation was performed - that it would have selected the negotiated version. A client MUST terminate the connection with a VERSION_NEGOTIATION_ERROR error code if the current QUIC version is not listed in the supported_versions list. A client MUST terminate with a VERSION_NEGOTIATION_ERROR error code if version negotiation occurred but it would have selected a different version based on the value of the supported_versions list.

When an endpoint accepts multiple QUIC versions, it can potentially interpret transport parameters as they are defined by any of the QUIC versions it supports. The version field in the QUIC packet header is authenticated using transport parameters. The position and the format of the version fields in transport parameters MUST either be identical across different QUIC versions, or be unambiguously different to ensure no confusion about their interpretation. One way that a new format could be introduced is to define a TLS extension with a different codepoint.

8. Address Validation

Address validation is used by QUIC to avoid being used for a traffic amplification attack. In such an attack, a packet is sent to a server with spoofed source address information that identifies a victim. If a server generates more or larger packets in response to that packet, the attacker can use the server to send more data toward the victim than it would be able to send on its own.

The primary defense against amplification attack is verifying that an endpoint is able to receive packets at the transport address that it claims. Address validation is performed both during connection establishment (see Section 8.1) and during connection migration (see Section 8.2).

8.1. Address Validation During Connection Establishment

Connection establishment implicitly provides address validation for both endpoints. In particular, receipt of a packet protected with Handshake keys confirms that the client received the Initial packet from the server. Once the server has successfully processed a Handshake packet from the client, it can consider the client address to have been validated.

Prior to validating the client address, servers MUST NOT send more than three times as many bytes as the number of bytes they have received. This limits the magnitude of any amplification attack that can be mounted using spoofed source addresses.

To ensure that the server is not overly constrained by this restriction, clients MUST send UDP datagrams with at least 1200 octets of payload until the server has completed address validation, see Section 14.

In order to prevent a handshake deadlock as a result of the server being unable to send, clients SHOULD send a packet upon a handshake timeout, as described in [QUIC-RECOVERY]. If the client has no data to retransmit and does not have Handshake keys, it SHOULD send an Initial packet in a UDP datagram of at least 1200 octets. If the client has Handshake keys, it SHOULD send a Handshake packet.

A server might wish to validate the client address before starting the cryptographic handshake. Client addresses can be verified using an address validation token. This token is delivered during connection establishment with a Retry packet (see Section 8.1.1) or in a previous connection using the NEW_TOKEN frame (see Section 8.1.2).

8.1.1. Address Validation using Retry Packets

QUIC uses token-based address validation during connection establishment. Any time the server wishes to validate a client address, it provides the client with a token. As long as it is not possible for an attacker to generate a valid token for its own address (see Section 8.1.3) and the client is able to return that token, it proves to the server that it received the token.

Upon receiving the client's Initial packet, the server can request address validation by sending a Retry packet (Section 17.7) containing a token. This token is repeated by the client in an Initial packet after it receives the Retry packet. In response to receiving a token in an Initial packet, a server can either abort the connection or permit it to proceed.

A server can also use a Retry packet to defer the state and processing costs of connection establishment. By giving the client a different connection ID to use, a server can cause the connection to be routed to a server instance with more resources available for new connections.

A flow showing the use of a Retry packet is shown in Figure 6.

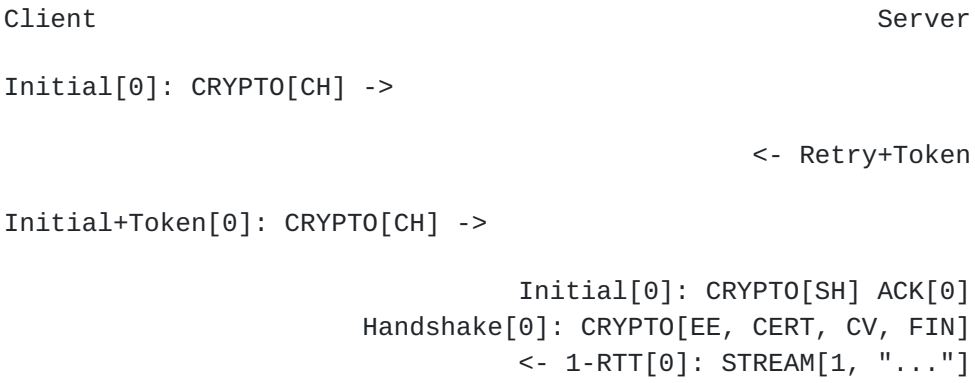


Figure 6: Example Handshake with Retry

8.1.2. Address Validation for Future Connections

A server MAY provide clients with an address validation token during one connection that can be used on a subsequent connection. Address validation is especially important with 0-RTT because a server potentially sends a significant amount of data to a client in response to 0-RTT data.

The server uses the NEW_TOKEN frame Section 19.18 to provide the client with an address validation token that can be used to validate future connections. The client may then use this token to validate future connections by including it in the Initial packet's header. The client MUST NOT use the token provided in a Retry for future connections.

Unlike the token that is created for a Retry packet, there might be some time between when the token is created and when the token is subsequently used. Thus, a resumption token SHOULD include an expiration time. The server MAY include either an explicit

expiration time or an issued timestamp and dynamically calculate the expiration time. It is also unlikely that the client port number is the same on two different connections; validating the port is therefore unlikely to be successful.

A resumption token SHOULD be constructed to be easily distinguishable from tokens that are sent in Retry packets as they are carried in the same field.

If the client has a token received in a NEW_TOKEN frame on a previous connection to what it believes to be the same server, it can include that value in the Token field of its Initial packet.

A token allows a server to correlate activity between the connection where the token was issued and any connection where it is used. Clients that want to break continuity of identity with a server MAY discard tokens provided using the NEW_TOKEN frame. Tokens obtained in Retry packets MUST NOT be discarded.

A client SHOULD NOT reuse a token. Reusing a token allows connections to be linked by entities on the network path (see Section 9.5). A client MUST NOT reuse a token if it believes that its point of network attachment has changed since the token was last used; that is, if there is a change in its local IP address or network interface. A client needs to start the connection process over if it migrates prior to completing the handshake.

When a server receives an Initial packet with an address validation token, it SHOULD attempt to validate it. If the token is invalid then the server SHOULD proceed as if the client did not have a validated address, including potentially sending a Retry. If the validation succeeds, the server SHOULD then allow the handshake to proceed.

Note: The rationale for treating the client as unvalidated rather than discarding the packet is that the client might have received the token in a previous connection using the NEW_TOKEN frame, and if the server has lost state, it might be unable to validate the token at all, leading to connection failure if the packet is discarded. A server MAY encode tokens provided with NEW_TOKEN frames and Retry packets differently, and validate the latter more strictly.

In a stateless design, a server can use encrypted and authenticated tokens to pass information to clients that the server can later recover and use to validate a client address. Tokens are not integrated into the cryptographic handshake and so they are not authenticated. For instance, a client might be able to reuse a

token. To avoid attacks that exploit this property, a server can limit its use of tokens to only the information needed validate client addresses.

8.1.3. Address Validation Token Integrity

An address validation token MUST be difficult to guess. Including a large enough random value in the token would be sufficient, but this depends on the server remembering the value it sends to clients.

A token-based scheme allows the server to offload any state associated with validation to the client. For this design to work, the token MUST be covered by integrity protection against modification or falsification by clients. Without integrity protection, malicious clients could generate or guess values for tokens that would be accepted by the server. Only the server requires access to the integrity protection key for tokens.

There is no need for a single well-defined format for the token because the server that generates the token also consumes it. A token could include information about the claimed client address (IP and port), a timestamp, and any other supplementary information the server will need to validate the token in the future.

8.2. Path Validation

Path validation is used during connection migration (see Section 9 and Section 9.6) by the migrating endpoint to verify reachability of a peer from a new local address. In path validation, endpoints test reachability between a specific local address and a specific peer address, where an address is the two-tuple of IP address and port.

Path validation tests that packets can be both sent to and received from a peer on the path. Importantly, it validates that the packets received from the migrating endpoint do not carry a spoofed source address.

Path validation can be used at any time by either endpoint. For instance, an endpoint might check that a peer is still in possession of its address after a period of quiescence.

Path validation is not designed as a NAT traversal mechanism. Though the mechanism described here might be effective for the creation of NAT bindings that support NAT traversal, the expectation is that one or other peer is able to receive packets without first having sent a packet on that path. Effective NAT traversal needs additional synchronization mechanisms that are not provided here.

An endpoint MAY bundle PATH_CHALLENGE and PATH_RESPONSE frames that are used for path validation with other frames. In particular, an endpoint may pad a packet carrying a PATH_CHALLENGE for PMTU discovery, or an endpoint may bundle a PATH_RESPONSE with its own PATH_CHALLENGE.

When probing a new path, an endpoint might want to ensure that its peer has an unused connection ID available for responses. The endpoint can send NEW_CONNECTION_ID and PATH_CHALLENGE frames in the same packet. This ensures that an unused connection ID will be available to the peer when sending a response.

8.3. Initiating Path Validation

To initiate path validation, an endpoint sends a PATH_CHALLENGE frame containing a random payload on the path to be validated.

An endpoint MAY send multiple PATH_CHALLENGE frames to guard against packet loss. An endpoint SHOULD NOT send a PATH_CHALLENGE more frequently than it would an Initial packet, ensuring that connection migration is no more load on a new path than establishing a new connection.

The endpoint MUST use fresh random data in every PATH_CHALLENGE frame so that it can associate the peer's response with the causative PATH_CHALLENGE.

8.4. Path Validation Responses

On receiving a PATH_CHALLENGE frame, an endpoint MUST respond immediately by echoing the data contained in the PATH_CHALLENGE frame in a PATH_RESPONSE frame. However, because a PATH_CHALLENGE might be sent from a spoofed address, an endpoint MUST limit the rate at which it sends PATH_RESPONSE frames and MAY silently discard PATH_CHALLENGE frames that would cause it to respond at a higher rate.

To ensure that packets can be both sent to and received from the peer, the PATH_RESPONSE MUST be sent on the same path as the triggering PATH_CHALLENGE. That is, from the same local address on which the PATH_CHALLENGE was received, to the same remote address from which the PATH_CHALLENGE was received.

8.5. Successful Path Validation

A new address is considered valid when a PATH_RESPONSE frame is received containing data that was sent in a previous PATH_CHALLENGE. Receipt of an acknowledgment for a packet containing a PATH_CHALLENGE

frame is not adequate validation, since the acknowledgment can be spoofed by a malicious peer.

For path validation to be successful, a PATH_RESPONSE frame MUST be received from the same remote address to which the corresponding PATH_CHALLENGE was sent. If a PATH_RESPONSE frame is received from a different remote address than the one to which the PATH_CHALLENGE was sent, path validation is considered to have failed, even if the data matches that sent in the PATH_CHALLENGE.

Additionally, the PATH_RESPONSE frame MUST be received on the same local address from which the corresponding PATH_CHALLENGE was sent. If a PATH_RESPONSE frame is received on a different local address than the one from which the PATH_CHALLENGE was sent, path validation is considered to have failed, even if the data matches that sent in the PATH_CHALLENGE. Thus, the endpoint considers the path to be valid when a PATH_RESPONSE frame is received on the same path with the same payload as the PATH_CHALLENGE frame.

8.6. Failed Path Validation

Path validation only fails when the endpoint attempting to validate the path abandons its attempt to validate the path.

Endpoints SHOULD abandon path validation based on a timer. When setting this timer, implementations are cautioned that the new path could have a longer round-trip time than the original.

Note that the endpoint might receive packets containing other frames on the new path, but a PATH_RESPONSE frame with appropriate data is required for path validation to succeed.

When an endpoint abandons path validation, it determines that the path is unusable. This does not necessarily imply a failure of the connection - endpoints can continue sending packets over other paths as appropriate. If no paths are available, an endpoint can wait for a new path to become available or close the connection.

A path validation might be abandoned for other reasons besides failure. Primarily, this happens if a connection migration to a new path is initiated while a path validation on the old path is in progress.

9. Connection Migration

The use of a connection ID allows connections to survive changes to endpoint addresses (that is, IP address and/or port), such as those

caused by an endpoint migrating to a new network. This section describes the process by which an endpoint migrates to a new address.

An endpoint MUST NOT initiate connection migration before the handshake is finished and the endpoint has 1-RTT keys. The design of QUIC relies on endpoints retaining a stable address for the duration of the handshake.

An endpoint also MUST NOT initiate connection migration if the peer sent the "disable_migration" transport parameter during the handshake. An endpoint which has sent this transport parameter, but detects that a peer has nonetheless migrated to a different network MAY treat this as a connection error of type INVALID_MIGRATION.

Not all changes of peer address are intentional migrations. The peer could experience NAT rebinding: a change of address due to a middlebox, usually a NAT, allocating a new outgoing port or even a new outgoing IP address for a flow. NAT rebinding is not connection migration as defined in this section, though an endpoint SHOULD perform path validation (Section 8.2) if it detects a change in the IP address of its peer.

This document limits migration of connections to new client addresses, except as described in Section 9.6. Clients are responsible for initiating all migrations. Servers do not send non-probing packets (see Section 9.1) toward a client address until they see a non-probing packet from that address. If a client receives packets from an unknown server address, the client MAY discard these packets.

9.1. Probing a New Path

An endpoint MAY probe for peer reachability from a new local address using path validation Section 8.2 prior to migrating the connection to the new local address. Failure of path validation simply means that the new path is not usable for this connection. Failure to validate a path does not cause the connection to end unless there are no valid alternative paths available.

An endpoint uses a new connection ID for probes sent from a new local address, see Section 9.5 for further discussion. An endpoint that uses a new local address needs to ensure that at least one new connection ID is available at the peer. That can be achieved by including a NEW_CONNECTION_ID frame in the probe.

Receiving a PATH_CHALLENGE frame from a peer indicates that the peer is probing for reachability on a path. An endpoint sends a PATH_RESPONSE in response as per Section 8.2.

PATH_CHALLENGE, PATH_RESPONSE, NEW_CONNECTION_ID, and PADDING frames are "probing frames", and all other frames are "non-probing frames". A packet containing only probing frames is a "probing packet", and a packet containing any other frame is a "non-probing packet".

9.2. Initiating Connection Migration

An endpoint can migrate a connection to a new local address by sending packets containing non-probing frames from that address.

Each endpoint validates its peer's address during connection establishment. Therefore, a migrating endpoint can send to its peer knowing that the peer is willing to receive at the peer's current address. Thus an endpoint can migrate to a new local address without first validating the peer's address.

When migrating, the new path might not support the endpoint's current sending rate. Therefore, the endpoint resets its congestion controller, as described in Section 9.4.

The new path might not have the same ECN capability. Therefore, the endpoint verifies ECN capability as described in Section 13.3.

Receiving acknowledgments for data sent on the new path serves as proof of the peer's reachability from the new address. Note that since acknowledgments may be received on any path, return reachability on the new path is not established. To establish return reachability on the new path, an endpoint MAY concurrently initiate path validation Section 8.2 on the new path.

9.3. Responding to Connection Migration

Receiving a packet from a new peer address containing a non-probing frame indicates that the peer has migrated to that address.

In response to such a packet, an endpoint MUST start sending subsequent packets to the new peer address and MUST initiate path validation (Section 8.2) to verify the peer's ownership of the unvalidated address.

An endpoint MAY send data to an unvalidated peer address, but it MUST protect against potential attacks as described in Section 9.3.1 and Section 9.3.2. An endpoint MAY skip validation of a peer address if that address has been seen recently.

An endpoint only changes the address that it sends packets to in response to the highest-numbered non-probing packet. This ensures

that an endpoint does not send packets to an old peer address in the case that it receives reordered packets.

After changing the address to which it sends non-probing packets, an endpoint could abandon any path validation for other addresses.

Receiving a packet from a new peer address might be the result of a NAT rebinding at the peer.

After verifying a new client address, the server SHOULD send new address validation tokens (Section 8) to the client.

9.3.1. Handling Address Spoofing by a Peer

It is possible that a peer is spoofing its source address to cause an endpoint to send excessive amounts of data to an unwilling host. If the endpoint sends significantly more data than the spoofing peer, connection migration might be used to amplify the volume of data that an attacker can generate toward a victim.

As described in Section 9.3, an endpoint is required to validate a peer's new address to confirm the peer's possession of the new address. Until a peer's address is deemed valid, an endpoint MUST limit the rate at which it sends data to this address. The endpoint MUST NOT send more than a minimum congestion window's worth of data per estimated round-trip time (kMinimumWindow, as defined in [QUIC-RECOVERY]). In the absence of this limit, an endpoint risks being used for a denial of service attack against an unsuspecting victim. Note that since the endpoint will not have any round-trip time measurements to this address, the estimate SHOULD be the default initial value (see [QUIC-RECOVERY]).

If an endpoint skips validation of a peer address as described in Section 9.3, it does not need to limit its sending rate.

9.3.2. Handling Address Spoofing by an On-path Attacker

An on-path attacker could cause a spurious connection migration by copying and forwarding a packet with a spoofed address such that it arrives before the original packet. The packet with the spoofed address will be seen to come from a migrating connection, and the original packet will be seen as a duplicate and dropped. After a spurious migration, validation of the source address will fail because the entity at the source address does not have the necessary cryptographic keys to read or respond to the PATH_CHALLENGE frame that is sent to it even if it wanted to.

To protect the connection from failing due to such a spurious migration, an endpoint MUST revert to using the last validated peer address when validation of a new peer address fails.

If an endpoint has no state about the last validated peer address, it MUST close the connection silently by discarding all connection state. This results in new packets on the connection being handled generically. For instance, an endpoint MAY send a stateless reset in response to any further incoming packets.

Note that receipt of packets with higher packet numbers from the legitimate peer address will trigger another connection migration. This will cause the validation of the address of the spurious migration to be abandoned.

9.4. Loss Detection and Congestion Control

The capacity available on the new path might not be the same as the old path. Packets sent on the old path SHOULD NOT contribute to congestion control or RTT estimation for the new path.

On confirming a peer's ownership of its new address, an endpoint SHOULD immediately reset the congestion controller and round-trip time estimator for the new path.

An endpoint MUST NOT return to the send rate used for the previous path unless it is reasonably sure that the previous send rate is valid for the new path. For instance, a change in the client's port number is likely indicative of a rebinding in a middlebox and not a complete change in path. This determination likely depends on heuristics, which could be imperfect; if the new path capacity is significantly reduced, ultimately this relies on the congestion controller responding to congestion signals and reducing send rates appropriately.

There may be apparent reordering at the receiver when an endpoint sends data and probes from/to multiple addresses during the migration period, since the two resulting paths may have different round-trip times. A receiver of packets on multiple paths will still send ACK frames covering all received packets.

While multiple paths might be used during connection migration, a single congestion control context and a single loss recovery context (as described in [QUIC-RECOVERY]) may be adequate. A sender can make exceptions for probe packets so that their loss detection is independent and does not unduly cause the congestion controller to reduce its sending rate. An endpoint might set a separate timer when a PATH_CHALLENGE is sent, which is cancelled when the corresponding

PATH_RESPONSE is received. If the timer fires before the PATH_RESPONSE is received, the endpoint might send a new PATH_CHALLENGE, and restart the timer for a longer period of time.

9.5. Privacy Implications of Connection Migration

Using a stable connection ID on multiple network paths allows a passive observer to correlate activity between those paths. An endpoint that moves between networks might not wish to have their activity correlated by any entity other than their peer, so different connection IDs are used when sending from different local addresses, as discussed in Section 5.1. For this to be effective endpoints need to ensure that connections IDs they provide cannot be linked by any other entity.

This eliminates the use of the connection ID for linking activity from the same connection on different networks. Protection of packet numbers ensures that packet numbers cannot be used to correlate activity. This does not prevent other properties of packets, such as timing and size, from being used to correlate activity.

Clients MAY move to a new connection ID at any time based on implementation-specific concerns. For example, after a period of network inactivity NAT rebinding might occur when the client begins sending data again.

A client might wish to reduce linkability by employing a new connection ID and source UDP port when sending traffic after a period of inactivity. Changing the UDP port from which it sends packets at the same time might cause the packet to appear as a connection migration. This ensures that the mechanisms that support migration are exercised even for clients that don't experience NAT rebindings or genuine migrations. Changing port number can cause a peer to reset its congestion state (see Section 9.4), so the port SHOULD only be changed infrequently.

Endpoints that use connection IDs with length greater than zero could have their activity correlated if their peers keep using the same destination connection ID after migration. Endpoints that receive packets with a previously unused Destination Connection ID SHOULD change to sending packets with a connection ID that has not been used on any other network path. The goal here is to ensure that packets sent on different paths cannot be correlated. To fulfill this privacy requirement, endpoints that initiate migration and use connection IDs with length greater than zero SHOULD provide their peers with new connection IDs before migration.

Caution: If both endpoints change connection ID in response to seeing a change in connection ID from their peer, then this can trigger an infinite sequence of changes.

9.6. Server's Preferred Address

QUIC allows servers to accept connections on one IP address and attempt to transfer these connections to a more preferred address shortly after the handshake. This is particularly useful when clients initially connect to an address shared by multiple servers but would prefer to use a unicast address to ensure connection stability. This section describes the protocol for migrating a connection to a preferred server address.

Migrating a connection to a new server address mid-connection is left for future work. If a client receives packets from a new server address not indicated by the preferred_address transport parameter, the client SHOULD discard these packets.

9.6.1. Communicating A Preferred Address

A server conveys a preferred address by including the preferred_address transport parameter in the TLS handshake.

Once the handshake is finished, the client SHOULD initiate path validation (see Section 8.2) of the server's preferred address using the connection ID provided in the preferred_address transport parameter.

If path validation succeeds, the client SHOULD immediately begin sending all future packets to the new server address using the new connection ID and discontinue use of the old server address. If path validation fails, the client MUST continue sending all future packets to the server's original IP address.

9.6.2. Responding to Connection Migration

A server might receive a packet addressed to its preferred IP address at any time after it accepts a connection. If this packet contains a PATH_CHALLENGE frame, the server sends a PATH_RESPONSE frame as per Section 8.2. The server MAY send other non-probing frames from its preferred address, but MUST continue sending all probing packets from its original IP address.

The server SHOULD also initiate path validation of the client using its preferred address and the address from which it received the client probe. This helps to guard against spurious migration initiated by an attacker.

Once the server has completed its path validation and has received a non-probing packet with a new largest packet number on its preferred address, the server begins sending non-probing packets to the client exclusively from its preferred IP address. It SHOULD drop packets for this connection received on the old IP address, but MAY continue to process delayed packets.

9.6.3. Interaction of Client Migration and Preferred Address

A client might need to perform a connection migration before it has migrated to the server's preferred address. In this case, the client SHOULD perform path validation to both the original and preferred server address from the client's new address concurrently.

If path validation of the server's preferred address succeeds, the client MUST abandon validation of the original address and migrate to using the server's preferred address. If path validation of the server's preferred address fails but validation of the server's original address succeeds, the client MAY migrate to its new address and continue sending to the server's original address.

If the connection to the server's preferred address is not from the same client address, the server MUST protect against potential attacks as described in Section 9.3.1 and Section 9.3.2. In addition to intentional simultaneous migration, this might also occur because the client's access network used a different NAT binding for the server's preferred address.

Servers SHOULD initiate path validation to the client's new address upon receiving a probe packet from a different address. Servers MUST NOT send more than a minimum congestion window's worth of non-probing packets to the new address before path validation is complete.

10. 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 three ways:

- idle timeout (Section 10.2)
- immediate close (Section 10.3)
- stateless reset (Section 10.4)

10.1. Closing and Draining Connection States

The closing and draining connection states exist to ensure that connections close cleanly and that delayed or reordered packets are properly discarded. These states SHOULD persist for three times the current Retransmission Timeout (RTO) interval as defined in [QUIC-RECOVERY].

An endpoint enters a closing period after initiating an immediate close (Section 10.3). While closing, an endpoint MUST NOT send packets unless they contain a CONNECTION_CLOSE or APPLICATION_CLOSE frame (see Section 10.3 for details). An endpoint retains only enough information to generate a packet containing a closing frame and to identify packets as belonging to the connection. The connection ID and QUIC version is sufficient information to identify packets for a closing connection; an endpoint can discard all other connection state. An endpoint MAY retain packet protection keys for incoming packets to allow it to read and process a closing frame.

The draining state is entered once an endpoint receives a signal that its peer is closing or draining. While otherwise identical to the closing state, an endpoint in the draining state MUST NOT send any packets. Retaining packet protection keys is unnecessary once a connection is in the draining state.

An endpoint MAY transition from the closing period to the draining period if it can confirm that its peer is also closing or draining. Receiving a closing frame is sufficient confirmation, as is receiving a stateless reset. The draining period SHOULD end when the closing period would have ended. In other words, the endpoint can use the same end time, but cease retransmission of the closing packet.

Disposing of connection state prior to the end of the closing or draining period could cause delayed or reordered packets to be handled poorly. Endpoints that have some alternative means to ensure that late-arriving packets on the connection do not create QUIC state, such as those that are able to close the UDP socket, MAY use an abbreviated draining period which can allow for faster resource recovery. Servers that retain an open socket for accepting new connections SHOULD NOT exit the closing or draining period early.

Once the closing or draining period has ended, an endpoint SHOULD discard all connection state. This results in new packets on the connection being handled generically. For instance, an endpoint MAY send a stateless reset in response to any further incoming packets.

The draining and closing periods do not apply when a stateless reset (Section 10.4) is sent.

An endpoint is not expected to handle key updates when it is closing or draining. A key update might prevent the endpoint from moving from the closing state to draining, but it otherwise has no impact.

An endpoint could receive packets from a new source address, indicating a client connection migration (Section 9), while in the closing period. An endpoint in the closing state MUST strictly limit the number of packets it sends to this new address until the address is validated (see Section 8.2). A server in the closing state MAY instead choose to discard packets received from a new source address.

10.2. Idle Timeout

If the idle timeout is enabled, a connection that remains idle for longer than the advertised idle timeout (see Section 18.1) is closed. A connection enters the draining state when the idle timeout expires.

Each endpoint advertises its own idle timeout to its peer. The idle timeout starts from the last packet received. In order to ensure that initiating new activity postpones an idle timeout, an endpoint restarts this timer when sending a packet. An endpoint does not postpone the idle timeout if another packet has been sent containing frames other than ACK or PADDING, and that other packet has not been acknowledged or declared lost. Packets that contain only ACK or PADDING frames are not acknowledged until an endpoint has other frames to send, so they could prevent the timeout from being refreshed.

The value for an idle timeout can be asymmetric. The value advertised by an endpoint is only used to determine whether the connection is live at that endpoint. An endpoint that sends packets near the end of the idle timeout period of a peer risks having those packets discarded if its peer enters the draining state before the packets arrive. If a peer could timeout within an RTO (see Section 4.3.3 of [QUIC-RECOVERY]), it is advisable to test for liveness before sending any data that cannot be retried safely.

10.3. Immediate Close

An endpoint sends a closing frame (CONNECTION_CLOSE or APPLICATION_CLOSE) to terminate the connection immediately. Any closing frame causes all streams to immediately become closed; open streams can be assumed to be implicitly reset.

After sending a closing frame, endpoints immediately enter the closing state. During the closing period, an endpoint that sends a closing frame SHOULD respond to any packet that it receives with another packet containing a closing frame. To minimize the state

that an endpoint maintains for a closing connection, endpoints MAY send the exact same packet. However, endpoints SHOULD limit the number of packets they generate containing a closing frame. For instance, an endpoint could progressively increase the number of packets that it receives before sending additional packets or increase the time between packets.

Note: Allowing retransmission of a packet contradicts other advice in this document that recommends the creation of new packet numbers for every packet. Sending new packet numbers is primarily of advantage to loss recovery and congestion control, which are not expected to be relevant for a closed connection. Retransmitting the final packet requires less state.

After receiving a closing frame, endpoints enter the draining state. An endpoint that receives a closing frame MAY send a single packet containing a closing frame before entering the draining state, using a CONNECTION_CLOSE frame and a NO_ERROR code if appropriate. An endpoint MUST NOT send further packets, which could result in a constant exchange of closing frames until the closing period on either peer ended.

An immediate close can be used after an application protocol has arranged to close a connection. This might be after the application protocols negotiates a graceful shutdown. The application protocol exchanges whatever messages that are needed to cause both endpoints to agree to close the connection, after which the application requests that the connection be closed. The application protocol can use an APPLICATION_CLOSE message with an appropriate error code to signal closure.

If the connection has been successfully established, endpoints MUST send any closing frames in a 1-RTT packet. Prior to connection establishment a peer might not have 1-RTT keys, so endpoints SHOULD send closing frames in a Handshake packet. If the endpoint does not have Handshake keys, or it is not certain that the peer has Handshake keys, it MAY send closing frames in an Initial packet. If multiple packets are sent, they can be coalesced (see Section 12.2) to facilitate retransmission.

10.4. Stateless Reset

A stateless reset is provided as an option of last resort for an endpoint that does not have access to the state of a connection. A crash or outage might result in peers continuing to send data to an endpoint that is unable to properly continue the connection. An endpoint that wishes to communicate a fatal connection error MUST use a closing frame if it has sufficient state to do so.

To support this process, a token is sent by endpoints. The token is carried in the NEW_CONNECTION_ID frame sent by either peer, and servers can specify the stateless_reset_token transport parameter during the handshake (clients cannot because their transport parameters don't have confidentiality protection). This value is protected by encryption, so only client and server know this value. Tokens sent via NEW_CONNECTION_ID frames are invalidated when their associated connection ID is retired via a RETIRE_CONNECTION_ID frame (Section 19.13).

An endpoint that receives packets that it cannot process sends a packet in the following layout:

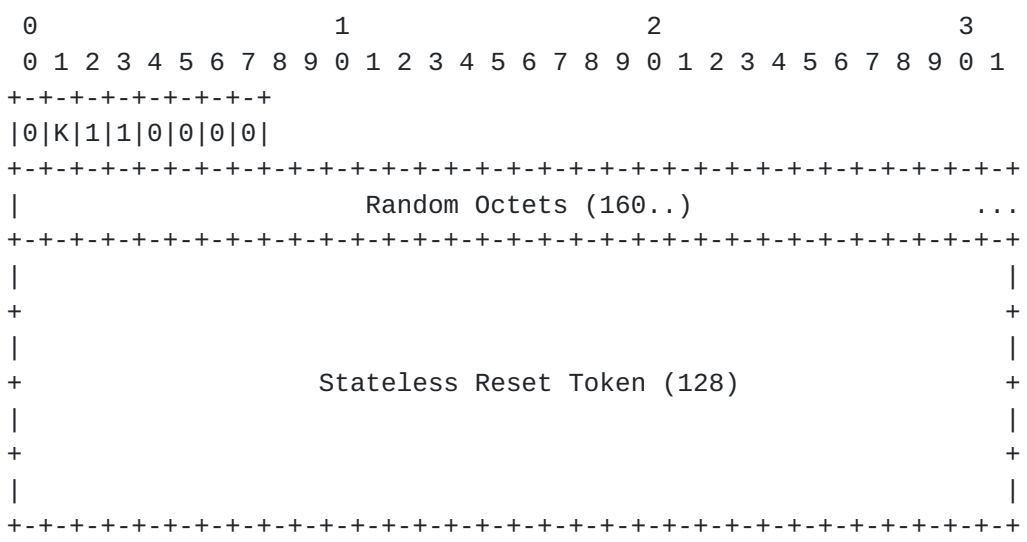


Figure 7: Stateless Reset Packet

This design ensures that a stateless reset packet is - to the extent possible - indistinguishable from a regular packet with a short header.

The message consists of a header octet, followed by an arbitrary number of random octets, followed by a Stateless Reset Token.

A stateless reset will be interpreted by a recipient as a packet with a short header. For the packet to appear as valid, the Random Octets field needs to include at least 20 octets of random or unpredictable values. This is intended to allow for a destination connection ID of the maximum length permitted, a packet number, and minimal payload. The Stateless Reset Token corresponds to the minimum expansion of the packet protection AEAD. More random octets might be necessary if the endpoint could have negotiated a packet protection scheme with a larger minimum AEAD expansion.

An endpoint SHOULD NOT send a stateless reset that is significantly larger than the packet it receives. Endpoints MUST discard packets that are too small to be valid QUIC packets. With the set of AEAD functions defined in [QUIC-TLS], packets less than 19 octets long are never valid.

An endpoint MAY send a stateless reset in response to a packet with a long header. This would not be effective if the stateless reset token was not yet available to a peer. In this QUIC version, packets with a long header are only used during connection establishment. Because the stateless reset token is not available until connection establishment is complete or near completion, ignoring an unknown packet with a long header might be more effective.

An endpoint cannot determine the Source Connection ID from a packet with a short header, therefore it cannot set the Destination Connection ID in the stateless reset packet. The Destination Connection ID will therefore differ from the value used in previous packets. A random Destination Connection ID makes the connection ID appear to be the result of moving to a new connection ID that was provided using a NEW_CONNECTION_ID frame (Section 19.12).

Using a randomized connection ID results in two problems:

- The packet might not reach the peer. If the Destination Connection ID is critical for routing toward the peer, then this packet could be incorrectly routed. This might also trigger another Stateless Reset in response, see Section 10.4.3. A Stateless Reset that is not correctly routed is ineffective in causing errors to be quickly detected and recovered. In this case, endpoints will need to rely on other methods - such as timers - to detect that the connection has failed.

- The randomly generated connection ID can be used by entities other than the peer to identify this as a potential stateless reset. An endpoint that occasionally uses different connection IDs might introduce some uncertainty about this.

Finally, the last 16 octets of the packet are set to the value of the Stateless Reset Token.

A stateless reset is not appropriate for signaling error conditions. An endpoint that wishes to communicate a fatal connection error MUST use a CONNECTION_CLOSE or APPLICATION_CLOSE frame if it has sufficient state to do so.

This stateless reset design is specific to QUIC version 1. An endpoint that supports multiple versions of QUIC needs to generate a

stateless reset that will be accepted by peers that support any version that the endpoint might support (or might have supported prior to losing state). Designers of new versions of QUIC need to be aware of this and either reuse this design, or use a portion of the packet other than the last 16 octets for carrying data.

10.4.1. Detecting a Stateless Reset

An endpoint detects a potential stateless reset when a packet with a short header either cannot be decrypted or is marked as a duplicate packet. The endpoint then compares the last 16 octets of the packet with the Stateless Reset Token provided by its peer, either in a NEW_CONNECTION_ID frame or the server's transport parameters. If these values are identical, the endpoint MUST enter the draining period and not send any further packets on this connection. If the comparison fails, the packet can be discarded.

10.4.2. Calculating a Stateless Reset Token

The stateless reset token MUST be difficult to guess. In order to create a Stateless Reset Token, an endpoint could randomly generate [RFC4086] a secret for every connection that it creates. However, this presents a coordination problem when there are multiple instances in a cluster or a storage problem for an endpoint that might lose state. Stateless reset specifically exists to handle the case where state is lost, so this approach is suboptimal.

A single static key can be used across all connections to the same endpoint by generating the proof using a second iteration of a preimage-resistant function that takes a static key and the connection ID chosen by the endpoint (see Section 5.1) as input. An endpoint could use HMAC [RFC2104] (for example, HMAC(static_key, connection_id)) or HKDF [RFC5869] (for example, using the static key as input keying material, with the connection ID as salt). The output of this function is truncated to 16 octets to produce the Stateless Reset Token for that connection.

An endpoint that loses state can use the same method to generate a valid Stateless Reset Token. The connection ID comes from the packet that the endpoint receives.

This design relies on the peer always sending a connection ID in its packets so that the endpoint can use the connection ID from a packet to reset the connection. An endpoint that uses this design MUST either use the same connection ID length for all connections or encode the length of the connection ID such that it can be recovered without state. In addition, it cannot provide a zero-length connection ID.

Revealing the Stateless Reset Token allows any entity to terminate the connection, so a value can only be used once. This method for choosing the Stateless Reset Token means that the combination of connection ID and static key cannot occur for another connection. A denial of service attack is possible if the same connection ID is used by instances that share a static key, or if an attacker can cause a packet to be routed to an instance that has no state but the same static key (see Section 21.8). A connection ID from a connection that is reset by revealing the Stateless Reset Token cannot be reused for new connections at nodes that share a static key.

Note that Stateless Reset packets do not have any cryptographic protection.

10.4.3. Looping

The design of a Stateless Reset is such that it is indistinguishable from a valid packet. This means that a Stateless Reset might trigger the sending of a Stateless Reset in response, which could lead to infinite exchanges.

An endpoint MUST ensure that every Stateless Reset that it sends is smaller than the packet which triggered it, unless it maintains state sufficient to prevent looping. In the event of a loop, this results in packets eventually being too small to trigger a response.

An endpoint can remember the number of Stateless Reset packets that it has sent and stop generating new Stateless Reset packets once a limit is reached. Using separate limits for different remote addresses will ensure that Stateless Reset packets can be used to close connections when other peers or connections have exhausted limits.

Reducing the size of a Stateless Reset below the recommended minimum size of 37 octets could mean that the packet could reveal to an observer that it is a Stateless Reset. Conversely, refusing to send a Stateless Reset in response to a small packet might result in Stateless Reset not being useful in detecting cases of broken connections where only very small packets are sent; such failures might only be detected by other means, such as timers.

An endpoint can increase the odds that a packet will trigger a Stateless Reset if it cannot be processed by padding it to at least 38 octets.

11. Error Handling

An endpoint that detects an error SHOULD signal the existence of that error to its peer. Both transport-level and application-level errors can affect an entire connection (see Section 11.1), while only application-level errors can be isolated to a single stream (see Section 11.2).

The most appropriate error code (Section 20) SHOULD be included in the frame that signals the error. Where this specification identifies error conditions, it also identifies the error code that is used.

A stateless reset (Section 10.4) is not suitable for any error that can be signaled with a CONNECTION_CLOSE, APPLICATION_CLOSE, or RST_STREAM frame. A stateless reset MUST NOT be used by an endpoint that has the state necessary to send a frame on the connection.

11.1. Connection Errors

Errors that result in the connection being unusable, such as an obvious violation of protocol semantics or corruption of state that affects an entire connection, MUST be signaled using a CONNECTION_CLOSE or APPLICATION_CLOSE frame (Section 19.3, Section 19.4). An endpoint MAY close the connection in this manner even if the error only affects a single stream.

Application protocols can signal application-specific protocol errors using the APPLICATION_CLOSE frame. Errors that are specific to the transport, including all those described in this document, are carried in a CONNECTION_CLOSE frame. Other than the type of error code they carry, these frames are identical in format and semantics.

A CONNECTION_CLOSE or APPLICATION_CLOSE frame could be sent in a packet that is lost. An endpoint SHOULD be prepared to retransmit a packet containing either frame type if it receives more packets on a terminated connection. Limiting the number of retransmissions and the time over which this final packet is sent limits the effort expended on terminated connections.

An endpoint that chooses not to retransmit packets containing CONNECTION_CLOSE or APPLICATION_CLOSE risks a peer missing the first such packet. The only mechanism available to an endpoint that continues to receive data for a terminated connection is to use the stateless reset process (Section 10.4).

An endpoint that receives an invalid CONNECTION_CLOSE or APPLICATION_CLOSE frame MUST NOT signal the existence of the error to its peer.

11.2. Stream Errors

If an application-level error affects a single stream, but otherwise leaves the connection in a recoverable state, the endpoint can send a RST_STREAM frame (Section 19.2) with an appropriate error code to terminate just the affected stream.

Other than STOPPING (Section 3.5), RST_STREAM MUST be instigated by the application and MUST carry an application error code. Resetting a stream without knowledge of the application protocol could cause the protocol to enter an unrecoverable state. Application protocols might require certain streams to be reliably delivered in order to guarantee consistent state between endpoints.

12. Packets and Frames

QUIC endpoints communicate by exchanging packets. Packets are carried in UDP datagrams (see Section 12.2) and have confidentiality and integrity protection (see Section 12.1).

This version of QUIC uses the long packet header (see Section 17.2) during connection establishment and the short header (see Section 17.3) once 1-RTT keys have been established.

Packets that carry the long header are Initial Section 17.5, Retry Section 17.7, Handshake Section 17.6, and 0-RTT Protected packets Section 12.1.

Packets with the short header are designed for minimal overhead and are used after a connection is established.

Version negotiation uses a packet with a special format (see Section 17.4).

12.1. Protected Packets

All QUIC packets except Version Negotiation and Retry packets use authenticated encryption with additional data (AEAD) [RFC5119] to provide confidentiality and integrity protection. Details of packet protection are found in [QUIC-TLS]; this section includes an overview of the process.

Initial packets are protected using keys that are statically derived. This packet protection is not effective confidentiality protection,

it only exists to ensure that the sender of the packet is on the network path. Any entity that receives the Initial packet from a client can recover the keys necessary to remove packet protection or to generate packets that will be successfully authenticated.

All other packets are protected with keys derived from the cryptographic handshake. The type of the packet from the long header or key phase from the short header are used to identify which encryption level - and therefore the keys - that are used. Packets protected with 0-RTT and 1-RTT keys are expected to have confidentiality and data origin authentication; the cryptographic handshake ensures that only the communicating endpoints receive the corresponding keys.

The packet number field contains a packet number, which has additional confidentiality protection that is applied after packet protection is applied (see [QUIC-TLS] for details). The underlying packet number increases with each packet sent, see Section 12.3 for details.

12.2. Coalescing Packets

A sender can coalesce multiple QUIC packets into one UDP datagram. This can reduce the number of UDP datagrams needed to complete the cryptographic handshake and starting sending data. Receivers MUST be able to process coalesced packets.

Coalescing packets in order of increasing encryption levels (Initial, 0-RTT, Handshake, 1-RTT) makes it more likely the receiver will be able to process all the packets in a single pass. A packet with a short header does not include a length, so it will always be the last packet included in a UDP datagram.

Senders MUST NOT coalesce QUIC packets for different connections into a single UDP datagram. Receivers SHOULD ignore any subsequent packets with a different Destination Connection ID than the first packet in the datagram.

Every QUIC packet that is coalesced into a single UDP datagram is separate and complete. Though the values of some fields in the packet header might be redundant, no fields are omitted. The receiver of coalesced QUIC packets MUST individually process each QUIC packet and separately acknowledge them, as if they were received as the payload of different UDP datagrams. For example, if decryption fails (because the keys are not available or any other reason) or the packet is of an unknown type, the receiver MAY either discard or buffer the packet for later processing and MUST attempt to process the remaining packets.

Retry packets (Section 17.7), Version Negotiation packets (Section 17.4), and packets with a short header cannot be followed by other packets in the same UDP datagram.

12.3. Packet Numbers

The packet number is an integer in the range 0 to 2^62-1. Where present, packet numbers are encoded as a variable-length integer (see Section 16). This number is used in determining the cryptographic nonce for packet protection. Each endpoint maintains a separate packet number for sending and receiving.

Version Negotiation (Section 17.4) and Retry Section 17.7 packets do not include a packet number.

Packet numbers are divided into 3 spaces in QUIC:

- Initial space: All Initial packets Section 17.5 are in this space.
- Handshake space: All Handshake packets Section 17.6 are in this space.
- Application data space: All 0-RTT and 1-RTT encrypted packets Section 12.1 are in this space.

As described in [QUIC-TLS], each packet type uses different protection keys.

Conceptually, a packet number space is the context in which a packet can be processed and acknowledged. Initial packets can only be sent with Initial packet protection keys and acknowledged in packets which are also Initial packets. Similarly, Handshake packets are sent at the Handshake encryption level and can only be acknowledged in Handshake packets.

This enforces cryptographic separation between the data sent in the different packet sequence number spaces. Each packet number space starts at packet number 0. Subsequent packets sent in the same packet number space MUST increase the packet number by at least one.

0-RTT and 1-RTT data exist in the same packet number space to make loss recovery algorithms easier to implement between the two packet types.

A QUIC endpoint MUST NOT reuse a packet number within the same packet number space in one connection (that is, under the same cryptographic keys). If the packet number for sending reaches 2^62 - 1, the sender MUST close the connection without sending a CONNECTION_CLOSE frame or

any further packets; an endpoint MAY send a Stateless Reset (Section 10.4) in response to further packets that it receives.

A receiver MUST discard a newly unprotected packet unless it is certain that it has not processed another packet with the same packet number from the same packet number space. Duplicate suppression MUST happen after removing packet protection for the reasons described in Section 9.3 of [QUIC-TLS]. An efficient algorithm for duplicate suppression can be found in Section 3.4.3 of [RFC2406].

Packet number encoding at a sender and decoding at a receiver are described in Section 17.1.

12.4. Frames and Frame Types

The payload of QUIC packets, after removing packet protection, commonly consists of a sequence of frames, as shown in Figure 8. Version Negotiation, Stateless Reset, and Retry packets do not contain frames.

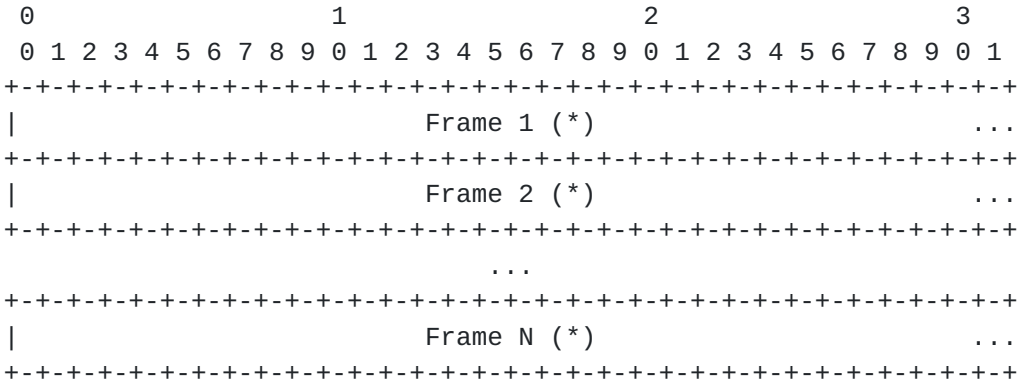


Figure 8: QUIC Payload

QUIC payloads MUST contain at least one frame, and 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, indicating its type, followed by additional type-dependent fields:

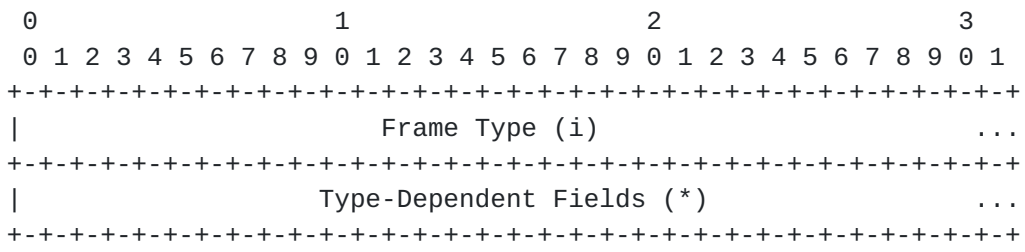


Figure 9: Generic Frame Layout

The frame types defined in this specification are listed in Table 3. The Frame Type in STREAM frames is used to carry other frame-specific flags. For all other frames, the Frame Type field simply identifies the frame. These frames are explained in more detail in Section 19.

Type Value	Frame Type Name	Definition
0x00	PADDING	Section 19.1
0x01	RST_STREAM	Section 19.2
0x02	CONNECTION_CLOSE	Section 19.3
0x03	APPLICATION_CLOSE	Section 19.4
0x04	MAX_DATA	Section 19.5
0x05	MAX_STREAM_DATA	Section 19.6
0x06	MAX_STREAM_ID	Section 19.7
0x07	PING	Section 19.8
0x08	BLOCKED	Section 19.9
0x09	STREAM_BLOCKED	Section 19.10
0x0a	STREAM_ID_BLOCKED	Section 19.11
0x0b	NEW_CONNECTION_ID	Section 19.12
0x0c	STOP_SENDING	Section 19.14
0x0d	RETIRE_CONNECTION_ID	Section 19.13
0x0e	PATH_CHALLENGE	Section 19.16
0x0f	PATH_RESPONSE	Section 19.17
0x10 - 0x17	STREAM	Section 19.19
0x18	CRYPTO	Section 19.20
0x19	NEW_TOKEN	Section 19.18
0x1a - 0x1b	ACK	Section 19.15

Table 3: Frame Types

All QUIC frames are idempotent. That is, a valid frame does not cause undesirable side effects or errors when received more than once.

The Frame Type field uses a variable length integer encoding (see Section 16) with one exception. To ensure simple and efficient implementations of frame parsing, a frame type MUST use the shortest possible encoding. Though a two-, four- or eight-octet encoding of the frame types defined in this document is possible, the Frame Type field for these frames is encoded on a single octet. For instance, though 0x4007 is a legitimate two-octet encoding for a variable-length integer with a value of 7, PING frames are always encoded as a single octet with the value 0x07. An endpoint MUST treat the receipt of a frame type that uses a longer encoding than necessary as a connection error of type PROTOCOL_VIOLATION.

13. Packetization and Reliability

A sender bundles one or more frames in a QUIC packet (see Section 12.4).

A sender can 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. An implementation may use knowledge about application sending behavior or heuristics to determine whether and for how long to wait. This waiting period is an implementation decision, and an implementation should be careful to delay conservatively, since any delay is likely to increase application-visible latency.

Stream multiplexing is achieved by interleaving STREAM frames from multiple streams into one or more QUIC packets. A single QUIC packet can include multiple STREAM frames from one or more streams.

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. Implementations are advised to bundle as few streams as necessary in outgoing packets without losing transmission efficiency to underfilled packets.

13.1. Packet Processing and Acknowledgment

A packet MUST NOT be acknowledged until packet protection has been successfully removed and all frames contained in the packet have been processed. For STREAM frames, this means the data has been enqueued in preparation to be received by the application protocol, but it does not require that data is delivered and consumed.

Once the packet has been fully processed, a receiver acknowledges receipt by sending one or more ACK frames containing the packet number of the received packet.

13.1.1. Sending ACK Frames

To avoid creating an indefinite feedback loop, an endpoint MUST NOT send an ACK frame in response to a packet containing only ACK or PADDING frames, even if there are packet gaps which precede the received packet. The endpoint MUST however acknowledge packets containing only ACK or PADDING frames when sending ACK frames in response to other packets.

While PADDING frames do not elicit an ACK frame from a receiver, they are considered to be in flight for congestion control purposes [QUIC-RECOVERY]. Sending only PADDING frames might cause the sender to become limited by the congestion controller (as described in [QUIC-RECOVERY]) with no acknowledgments forthcoming from the receiver. Therefore, a sender should ensure that other frames are sent in addition to PADDING frames to elicit acknowledgments from the receiver.

An endpoint MUST NOT send more than one packet containing only an ACK frame per received packet that contains frames other than ACK and PADDING frames.

The receiver's delayed acknowledgment timer SHOULD NOT exceed the current RTT estimate or the value it indicates in the "max_ack_delay" transport parameter. This ensures an acknowledgment is sent at least once per RTT when packets needing acknowledgement are received. The sender can use the receiver's "max_ack_delay" value in determining timeouts for timer-based retransmission.

Strategies and implications of the frequency of generating acknowledgments are discussed in more detail in [QUIC-RECOVERY].

To limit ACK Blocks to those that have not yet been received by the sender, the receiver SHOULD track which ACK frames have been acknowledged by its peer. Once an ACK frame has been acknowledged, the packets it acknowledges SHOULD NOT be acknowledged again.

Because ACK frames are not sent in response to ACK-only packets, a receiver that is only sending ACK frames will only receive acknowledgements for its packets if the sender includes them in packets with non-ACK frames. A sender SHOULD bundle ACK frames with other frames when possible.

To limit receiver state or the size of ACK frames, a receiver MAY limit the number of ACK Blocks it sends. A receiver can do this even without receiving acknowledgment of its ACK frames, with the knowledge this could cause the sender to unnecessarily retransmit some data. Standard QUIC [QUIC-RECOVERY] algorithms declare packets lost after sufficiently newer packets are acknowledged. Therefore, the receiver SHOULD repeatedly acknowledge newly received packets in preference to packets received in the past.

13.1.2. ACK Frames and Packet Protection

ACK frames MUST only be carried in a packet that has the same packet number space as the packet being ACKed (see Section 12.1). For instance, packets that are protected with 1-RTT keys MUST be acknowledged in packets that are also protected with 1-RTT keys.

Packets that a client sends with 0-RTT packet protection MUST be acknowledged by the server in packets protected by 1-RTT keys. This can mean that the client is unable to use these acknowledgments if the server cryptographic handshake messages are delayed or lost. Note that the same limitation applies to other data sent by the server protected by the 1-RTT keys.

Endpoints SHOULD send acknowledgments for packets containing CRYPTO frames with a reduced delay; see Section 4.3.1 of [QUIC-RECOVERY].

13.2. Retransmission of Information

QUIC packets that are determined to be lost are not retransmitted whole. The same applies to the frames that are contained within lost packets. Instead, the information that might be carried in frames is sent again in new frames as needed.

New frames and packets are used to carry information that is determined to have been lost. In general, information is sent again when a packet containing that information is determined to be lost and sending ceases when a packet containing that information is acknowledged.

- Data sent in CRYPTO frames is retransmitted according to the rules in [QUIC-RECOVERY], until all data has been acknowledged.

- Application data sent in STREAM frames is retransmitted in new STREAM frames unless the endpoint has sent a RST_STREAM for that stream. Once an endpoint sends a RST_STREAM frame, no further STREAM frames are needed.

- The most recent set of acknowledgments are sent in ACK frames. An ACK frame SHOULD contain all unacknowledged acknowledgments, as described in Section 13.1.1.

- Cancellation of stream transmission, as carried in a RST_STREAM frame, is sent until acknowledged or until all stream data is acknowledged by the peer (that is, either the "Reset Recvd" or "Data Recvd" state is reached on the send stream). The content of a RST_STREAM frame MUST NOT change when it is sent again.

- Similarly, a request to cancel stream transmission, as encoded in a STOP_SENDING frame, is sent until the receive stream enters either a "Data Recvd" or "Reset Recvd" state, see Section 3.5.

- Connection close signals, including those that use CONNECTION_CLOSE and APPLICATION_CLOSE frames, are not sent again when packet loss is detected, but as described in Section 10.

- The current connection maximum data is sent in MAX_DATA frames. An updated value is sent in a MAX_DATA frame if the packet containing the most recently sent MAX_DATA frame is declared lost, or when the endpoint decides to update the limit. Care is necessary to avoid sending this frame too often as the limit can increase frequently and cause an unnecessarily large number of MAX_DATA frames to be sent.

- The current maximum stream data offset is sent in MAX_STREAM_DATA frames. Like MAX_DATA, an updated value is sent when the packet containing the most recent MAX_STREAM_DATA frame for a stream is lost or when the limit is updated, with care taken to prevent the frame from being sent too often. An endpoint SHOULD stop sending MAX_STREAM_DATA frames when the receive stream enters a "Size Known" state.

- The maximum stream ID for a stream of a given type is sent in MAX_STREAM_ID frames. Like MAX_DATA, an updated value is sent when a packet containing the most recent MAX_STREAM_ID for a stream type frame is declared lost or when the limit is updated, with care taken to prevent the frame from being sent too often.

- Blocked signals are carried in BLOCKED, STREAM_BLOCKED, and STREAM_ID_BLOCKED frames. BLOCKED streams have connection scope, STREAM_BLOCKED frames have stream scope, and STREAM_ID_BLOCKED

frames are scoped to a specific stream type. New frames are sent if packets containing the most recent frame for a scope is lost, but only while the endpoint is blocked on the corresponding limit. These frames always include the limit that is causing blocking at the time that they are transmitted.

- A liveness or path validation check using PATH_CHALLENGE frames is sent periodically until a matching PATH_RESPONSE frame is received or until there is no remaining need for liveness or path validation checking. PATH_CHALLENGE frames include a different payload each time they are sent.

- Responses to path validation using PATH_RESPONSE frames are sent just once. A new PATH_CHALLENGE frame will be sent if another PATH_RESPONSE frame is needed.

- New connection IDs are sent in NEW_CONNECTION_ID frames and retransmitted if the packet containing them is lost. Retransmissions of this frame carry the same sequence number value. Likewise, retired connection IDs are sent in RETIRE_CONNECTION_ID frames and retransmitted if the packet containing them is lost.

- PADDING frames contain no information, so lost PADDING frames do not require repair.

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

13.3. Explicit Congestion Notification

QUIC endpoints use Explicit Congestion Notification (ECN) [RFC3168] to detect and respond to network congestion. ECN allows a network node to indicate congestion in the network by setting a codepoint in the IP header of a packet instead of dropping it. Endpoints react to congestion by reducing their sending rate in response, as described in [QUIC-RECOVERY].

To use ECN, QUIC endpoints first determine whether a path supports ECN marking and the peer is able to access the ECN codepoint in the IP header. A network path does not support ECN if ECN marked packets get dropped or ECN markings are rewritten on the path. An endpoint verifies the path, both during connection establishment and when migrating to a new path (see Section 9).

13.3.1. ECN Counters

On receiving a packet with an ECT or CE codepoint, an endpoint that can access the IP ECN codepoints increases the corresponding ECT(0), ECT(1), or CE count, and includes these counters in subsequent (see Section 13.1) ACK frames (see Section 19.15).

A packet detected by a receiver as a duplicate does not affect the receiver's local ECN codepoint counts; see (Section 21.7) for relevant security concerns.

If an endpoint receives a packet without an ECT or CE codepoint, it responds per Section 13.1 with an ACK frame. If an endpoint does not have access to received ECN codepoints, it acknowledges received packets per Section 13.1 with an ACK frame.

13.3.2. ECN Verification

Each endpoint independently verifies and enables use of ECN by setting the IP header ECN codepoint to ECN Capable Transport (ECT) for the path from it to the other peer. Even if ECN is not used on the path to the peer, the endpoint MUST provide feedback about ECN markings received (if accessible).

To verify both that a path supports ECN and the peer can provide ECN feedback, an endpoint MUST set the ECT(0) codepoint in the IP header of all outgoing packets [RFC8311].

If an ECT codepoint set in the IP header is not corrupted by a network device, then a received packet contains either the codepoint sent by the peer or the Congestion Experienced (CE) codepoint set by a network device that is experiencing congestion.

If a packet sent with an ECT codepoint is newly acknowledged by the peer in an ACK frame without ECN feedback, the endpoint stops setting ECT codepoints in subsequent packets, with the expectation that either the network or the peer no longer supports ECN.

To protect the connection from arbitrary corruption of ECN codepoints by the network, an endpoint verifies the following when an ACK frame is received:

- The increase in ECT(0) and ECT(1) counters MUST be at least the number of packets newly acknowledged that were sent with the corresponding codepoint.

- The total increase in ECT(0), ECT(1), and CE counters reported in the ACK frame MUST be at least the total number of packets newly acknowledged in this ACK frame.

An endpoint could miss acknowledgements for a packet when ACK frames are lost. It is therefore possible for the total increase in ECT(0), ECT(1), and CE counters to be greater than the number of packets acknowledged in an ACK frame. When this happens, the local reference counts MUST be increased to match the counters in the ACK frame.

Upon successful verification, an endpoint continues to set ECT codepoints in subsequent packets with the expectation that the path is ECN-capable.

If verification fails, then the endpoint ceases setting ECT codepoints in subsequent packets with the expectation that either the network or the peer does not support ECN.

If an endpoint sets ECT codepoints on outgoing packets and encounters a retransmission timeout due to the absence of acknowledgments from the peer (see [QUIC-RECOVERY]), or if an endpoint has reason to believe that a network element might be corrupting ECN codepoints, the endpoint MAY cease setting ECT codepoints in subsequent packets. Doing so allows the connection to traverse network elements that drop or corrupt ECN codepoints in the IP header.

14. Packet Size

The QUIC packet size includes the QUIC header and integrity check, but not the UDP or IP header.

Clients MUST ensure that the first Initial packet they send is sent in a UDP datagram that is at least 1200 octets. Padding the Initial packet or including a 0-RTT packet in the same datagram are ways to meet this requirement. Sending a UDP datagram of this size ensures that the network path supports a reasonable Maximum Transmission Unit (MTU), and helps reduce the amplitude of amplification attacks caused by server responses toward an unverified client address, see Section 8.

The payload of a UDP datagram carrying the Initial packet MUST be expanded to at least 1200 octets, by adding PADDING frames to the Initial packet and/or by combining the Initial packet with a 0-RTT packet (see Section 12.2).

The datagram containing the first Initial packet from a client MAY exceed 1200 octets if the client believes that the Path Maximum Transmission Unit (PMTU) supports the size that it chooses.

A server MAY send a CONNECTION_CLOSE frame with error code PROTOCOL_VIOLATION in response to the first Initial packet it receives from a client if the UDP datagram is smaller than 1200 octets. It MUST NOT send any other frame type in response, or otherwise behave as if any part of the offending packet was processed as valid.

The server MUST also limit the number of bytes it sends before validating the address of the client, see Section 8.

14.1. Path Maximum Transmission Unit

The Path Maximum Transmission Unit (PMTU) is the maximum size of the entire IP header, UDP header, and UDP payload. The UDP payload includes the QUIC packet header, protected payload, and any authentication fields.

All QUIC packets SHOULD be sized to fit within the estimated PMTU to avoid IP fragmentation or packet drops. To optimize bandwidth efficiency, endpoints SHOULD use Packetization Layer PMTU Discovery ([PLPMTUD]). Endpoints MAY use PMTU Discovery ([PMTUDv4], [PMTUDv6]) for detecting the PMTU, setting the PMTU appropriately, and storing the result of previous PMTU determinations.

In the absence of these mechanisms, QUIC endpoints SHOULD NOT send IP packets larger than 1280 octets. Assuming the minimum IP header size, this results in a QUIC packet size of 1232 octets for IPv6 and 1252 octets for IPv4. Some QUIC implementations MAY be more conservative in computing allowed QUIC packet size given unknown tunneling overheads or IP header options.

QUIC endpoints that implement any kind of PMTU discovery SHOULD maintain an estimate for each combination of local and remote IP addresses. Each pairing of local and remote addresses could have a different maximum MTU in the path.

QUIC depends on the network path supporting an MTU of at least 1280 octets. This is the IPv6 minimum MTU and therefore also supported by most modern IPv4 networks. An endpoint MUST NOT reduce its MTU below this number, even if it receives signals that indicate a smaller limit might exist.

If a QUIC endpoint determines that the PMTU between any pair of local and remote IP addresses has fallen below 1280 octets, it MUST immediately cease sending QUIC packets on the affected path. This could result in termination of the connection if an alternative path cannot be found.

14.1.1. IPv4 PMTU Discovery

Traditional ICMP-based path MTU discovery in IPv4 [PMTUDv4] is potentially vulnerable to off-path attacks that successfully guess the IP/port 4-tuple and reduce the MTU to a bandwidth-inefficient value. TCP connections mitigate this risk by using the (at minimum) 8 bytes of transport header echoed in the ICMP message to validate the TCP sequence number as valid for the current connection. However, as QUIC operates over UDP, in IPv4 the echoed information could consist only of the IP and UDP headers, which usually has insufficient entropy to mitigate off-path attacks.

As a result, endpoints that implement PMTUD in IPv4 SHOULD take steps to mitigate this risk. For instance, an application could:

- Set the IPv4 Don't Fragment (DF) bit on a small proportion of packets, so that most invalid ICMP messages arrive when there are no DF packets outstanding, and can therefore be identified as spurious.

- Store additional information from the IP or UDP headers from DF packets (for example, the IP ID or UDP checksum) to further authenticate incoming Datagram Too Big messages.

- Any reduction in PMTU due to a report contained in an ICMP packet is provisional until QUIC's loss detection algorithm determines that the packet is actually lost.

14.2. Special Considerations for Packetization Layer PMTU Discovery

The PADDING frame provides a useful option for PMTU probe packets. PADDING frames generate acknowledgements, but they need not be delivered reliably. As a result, the loss of PADDING frames in probe packets does not require delay-inducing retransmission. However, PADDING frames do consume congestion window, which may delay the transmission of subsequent application data.

When implementing the algorithm in Section 7.2 of [PLPMTUD], the initial value of search_low SHOULD be consistent with the IPv6 minimum packet size. Paths that do not support this size cannot deliver Initial packets, and therefore are not QUIC-compliant.

Section 7.3 of [PLPMTUD] discusses trade-offs between small and large increases in the size of probe packets. As QUIC probe packets need not contain application data, aggressive increases in probe size carry fewer consequences.

15. Versions

QUIC versions are identified using a 32-bit unsigned number.

The version 0x00000000 is reserved to represent version negotiation. This version of the specification is identified by the number 0x00000001.

Other versions of QUIC might have different properties to this version. The properties of QUIC that are guaranteed to be consistent across all versions of the protocol are described in [QUIC-INVARIANTS].

Version 0x00000001 of QUIC uses TLS as a cryptographic handshake protocol, as described in [QUIC-TLS].

Versions with the most significant 16 bits of the version number cleared are reserved for use in future IETF consensus documents.

Versions that follow the pattern 0x?a?a?a?a are reserved for use in forcing version negotiation to be exercised. That is, any version number where the low four bits of all octets is 1010 (in binary). A client or server MAY advertise support for any of these reserved versions.

Reserved version numbers will probably never represent a real protocol; a client MAY use one of these version numbers with the expectation that the server will initiate version negotiation; a server MAY advertise support for one of these versions and can expect that clients ignore the value.

[[RFC editor: please remove the remainder of this section before publication.]]

The version number for the final version of this specification (0x00000001), is reserved for the version of the protocol that is published as an RFC.

Version numbers used to identify IETF drafts are created by adding the draft number to 0xff000000. For example, draft-ietf-quic-transport-13 would be identified as 0xff00000D.

Implementors are encouraged to register version numbers of QUIC that they are using for private experimentation on the GitHub wiki at <https://github.com/quicwg/base-drafts/wiki/QUIC-Versions>.

16. Variable-Length Integer Encoding

QUIC packets and frames commonly use a variable-length encoding for non-negative integer values. This encoding ensures that smaller integer values need fewer octets to encode.

The QUIC variable-length integer encoding reserves the two most significant bits of the first octet to encode the base 2 logarithm of the integer encoding length in octets. The integer value is encoded on the remaining bits, in network byte order.

This means that integers are encoded on 1, 2, 4, or 8 octets and can encode 6, 14, 30, or 62 bit values respectively. Table 4 summarizes the encoding properties.

2Bit	Length	Usable Bits	Range
00	1	6	0-63
01	2	14	0-16383
10	4	30	0-1073741823
11	8	62	0-4611686018427387903

Table 4: Summary of Integer Encodings

For example, the eight octet sequence c2 19 7c 5e ff 14 e8 8c (in hexadecimal) decodes to the decimal value 151288809941952652; the four octet sequence 9d 7f 3e 7d decodes to 494878333; the two octet sequence 7b bd decodes to 15293; and the single octet 25 decodes to 37 (as does the two octet sequence 40 25).

Error codes (Section 20) and versions Section 15 are described using integers, but do not use this encoding.

17. Packet Formats

All numeric values are encoded in network byte order (that is, big-endian) and all field sizes are in bits. Hexadecimal notation is used for describing the value of fields.

17.1. Packet Number Encoding and Decoding

Packet numbers in long and short packet headers are encoded as follows. The number of bits required to represent the packet number is first reduced by including only a variable number of the least significant bits of the packet number. One or two of the most significant bits of the first octet are then used to represent how many bits of the packet number are provided, as shown in Table 5.

First octet pattern	Encoded Length	Bits Present
0b0xxxxxxx	1 octet	7
0b10xxxxxx	2	14
0b11xxxxxx	4	30

Table 5: Packet Number Encodings for Packet Headers

Note that these encodings are similar to those in Section 16, but use different values.

Finally, the encoded packet number is protected as described in Section 5.3 of [QUIC-TLS].

The sender MUST use a packet number size able to represent more than twice as large a range than the difference between the largest acknowledged packet and packet number being sent. A peer receiving the packet will then correctly decode the packet number, unless the packet is delayed in transit such that it arrives after many higher-numbered packets have been received. An endpoint SHOULD use a large enough packet number encoding to allow the packet number to be recovered even if the packet arrives after packets that are sent afterwards.

As a result, the size of the packet number encoding is at least one more than the base 2 logarithm of the number of contiguous unacknowledged packet numbers, including the new packet.

For example, if an endpoint has received an acknowledgment for packet 0x6afa2f, sending a packet with a number of 0x6b2d79 requires a packet number encoding with 14 bits or more; whereas the 30-bit packet number encoding is needed to send a packet with a number of 0x6bc107.

At a receiver, protection of the packet number is removed prior to recovering the full packet number. The full packet number is then reconstructed based on the number of significant bits present, the value of those bits, and the largest packet number received on a successfully authenticated packet. Recovering the full packet number is necessary to successfully remove packet protection.

Once packet number protection is removed, the packet number is decoded by finding the packet number value that is closest to the next expected packet. The next expected packet is the highest received packet number plus one. For example, if the highest successfully authenticated packet had a packet number of 0xaa82f30e, then a packet containing a 14-bit value of 0x9b3 will be decoded as 0xaa8309b3. Example pseudo-code for packet number decoding can be found in Appendix A.

17.2. Long Header Packet

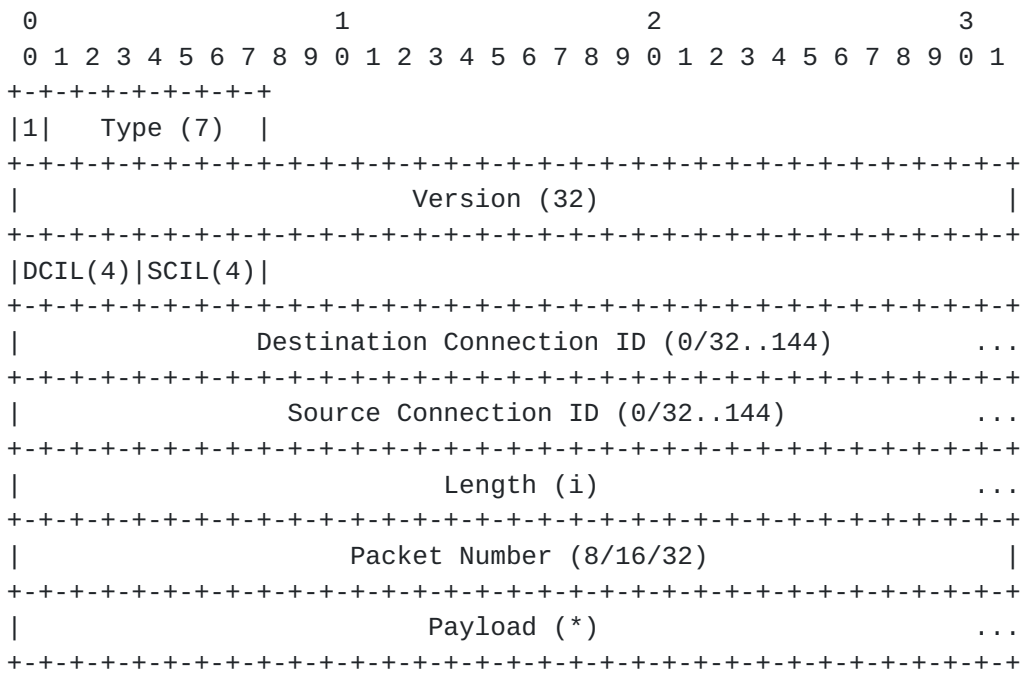


Figure 10: Long Header Packet Format

Long headers are used for packets that are sent prior to the completion of version negotiation and establishment of 1-RTT keys. Once both conditions are met, a sender switches to sending packets using the short header (Section 17.3). The long form allows for special packets - such as the Version Negotiation packet - to be represented in this uniform fixed-length packet format. Packets that use the long header contain the following fields:

Header Form: The most significant bit (0x80) of octet 0 (the first octet) is set to 1 for long headers.

Long Packet Type: The remaining seven bits of octet 0 contain the packet type. This field can indicate one of 128 packet types. The types specified for this version are listed in Table 6.

Version: The QUIC Version is a 32-bit field that follows the Type. This field indicates which version of QUIC is in use and determines how the rest of the protocol fields are interpreted.

DCIL and SCIL: The octet following the version contains the lengths of the two connection ID fields that follow it. These lengths are encoded as two 4-bit unsigned integers. The Destination Connection ID Length (DCIL) field occupies the 4 high bits of the octet and the Source Connection ID Length (SCIL) field occupies the 4 low bits of the octet. An encoded length of 0 indicates that the connection ID is also 0 octets in length. Non-zero encoded lengths are increased by 3 to get the full length of the connection ID, producing a length between 4 and 18 octets inclusive. For example, an octet with the value 0x50 describes an 8-octet Destination Connection ID and a zero-length Source Connection ID.

Destination Connection ID: The Destination Connection ID field follows the connection ID lengths and is either 0 octets in length or between 4 and 18 octets. Section 7.2 describes the use of this field in more detail.

Source Connection ID: The Source Connection ID field follows the Destination Connection ID and is either 0 octets in length or between 4 and 18 octets. Section 7.2 describes the use of this field in more detail.

Length: The length of the remainder of the packet (that is, the Packet Number and Payload fields) in octets, encoded as a variable-length integer (Section 16).

Packet Number: The packet number field is 1, 2, or 4 octets long. The packet number has confidentiality protection separate from packet protection, as described in Section 5.3 of [QUIC-TLS]. The length of the packet number field is encoded in the plaintext packet number. See Section 17.1 for details.

Payload: The payload of the packet.

The following packet types are defined:

Type	Name	Section
0x7F	Initial	Section 17.5
0x7E	Retry	Section 17.7
0x7D	Handshake	Section 17.6
0x7C	0-RTT Protected	Section 12.1

Table 6: Long Header Packet Types

The header form, type, connection ID lengths octet, destination and source connection IDs, and version fields of a long header packet are version-independent. The packet number and values for packet types defined in Table 6 are version-specific. See [QUIC-INVARIANTS] for details on how packets from different versions of QUIC are interpreted.

The interpretation of the fields and the payload are specific to a version and packet type. Type-specific semantics for this version are described in the following sections.

The end of the packet is determined by the Length field. The Length field covers both the Packet Number and Payload fields, both of which are confidentiality protected and initially of unknown length. The size of the Payload field is learned once the packet number protection is removed. The Length field enables packet coalescing (Section 12.2).

17.3. Short Header Packet

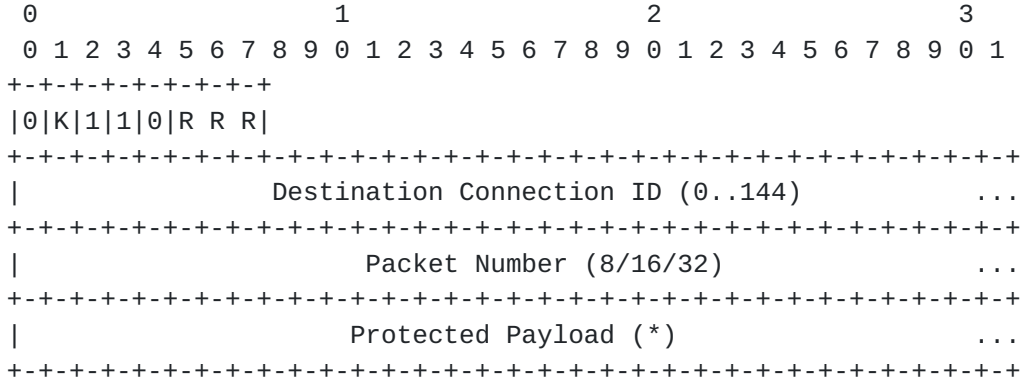


Figure 11: Short Header Packet Format

The short header can be used after the version and 1-RTT keys are negotiated. Packets that use the short header contain the following fields:

Header Form: The most significant bit (0x80) of octet 0 is set to 0 for the short header.

Key Phase Bit: The second bit (0x40) of octet 0 indicates the key phase, which allows a recipient of a packet to identify the packet protection keys that are used to protect the packet. See [QUIC-TLS] for details.

[[Editor's Note: this section should be removed and the bit definitions changed before this draft goes to the IESG.]]

Third Bit: The third bit (0x20) of octet 0 is set to 1.

[[Editor's Note: this section should be removed and the bit definitions changed before this draft goes to the IESG.]]

Fourth Bit: The fourth bit (0x10) of octet 0 is set to 1.

[[Editor's Note: this section should be removed and the bit definitions changed before this draft goes to the IESG.]]

Google QUIC Demultiplexing Bit: The fifth bit (0x8) of octet 0 is set to 0. This allows implementations of Google QUIC to distinguish Google QUIC packets from short header packets sent by a client because Google QUIC servers expect the connection ID to always be present. The special interpretation of this bit SHOULD be removed from this specification when Google QUIC has finished transitioning to the new header format.

Reserved: The sixth, seventh, and eighth bits (0x7) of octet 0 are reserved for experimentation. Endpoints MUST ignore these bits on packets they receive unless they are participating in an experiment that uses these bits. An endpoint not actively using these bits SHOULD set the value randomly on packets they send to protect against unwanted inference about particular values.

Destination Connection ID: The Destination Connection ID is a connection ID that is chosen by the intended recipient of the packet. See Section 5.1 for more details.

Packet Number: The packet number field is 1, 2, or 4 octets long. The packet number has confidentiality protection separate from packet protection, as described in Section 5.3 of [QUIC-TLS]. The

length of the packet number field is encoded in the plaintext packet number. See Section 17.1 for details.

Protected Payload: Packets with a short header always include a 1-RTT protected payload.

The header form and connection ID field of a short header packet are version-independent. The remaining fields are specific to the selected QUIC version. See [QUIC-INVARIANTS] for details on how packets from different versions of QUIC are interpreted.

17.4. Version Negotiation Packet

A Version Negotiation packet is inherently not version-specific, and does not use the long packet header (see Section 17.2). Upon receipt by a client, it will appear to be a packet using the long header, but will be identified as a Version Negotiation packet based on the Version field having a value of 0.

The Version Negotiation packet is a response to a client packet that contains a version that is not supported by the server, and is only sent by servers.

The layout of a Version Negotiation packet is:

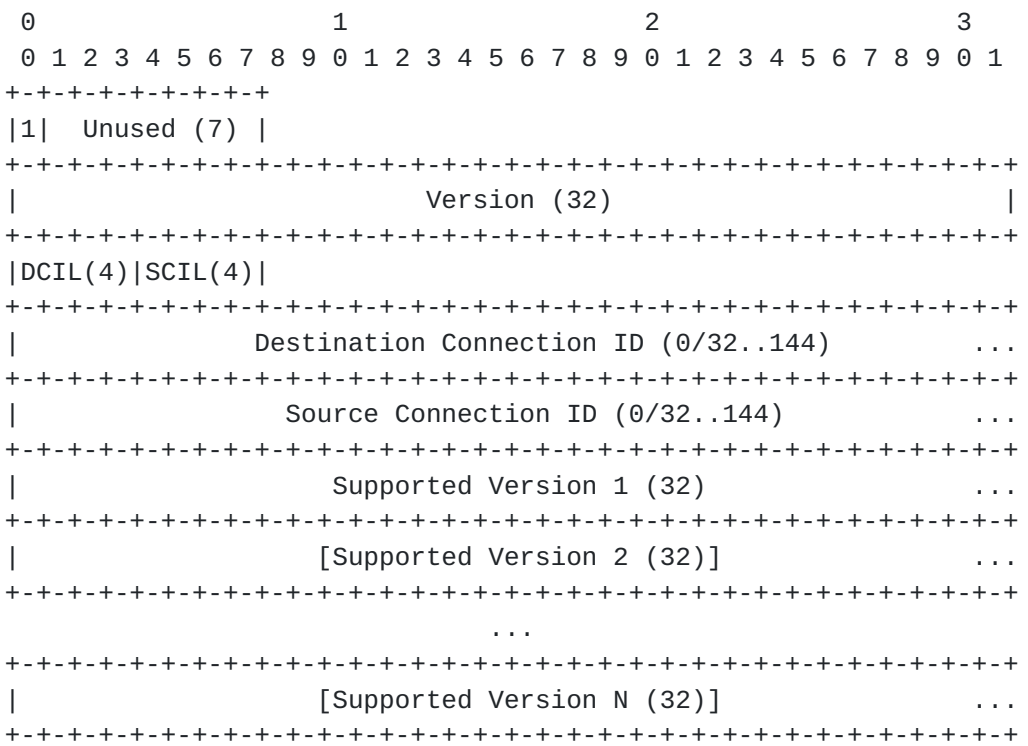


Figure 12: Version Negotiation Packet

The value in the Unused field is selected randomly by the server.

The Version field of a Version Negotiation packet MUST be set to 0x00000000.

The server MUST include the value from the Source Connection ID field of the packet it receives in the Destination Connection ID field. The value for Source Connection ID MUST be copied from the Destination Connection ID of the received packet, which is initially randomly selected by a client. Echoing both connection IDs gives clients some assurance that the server received the packet and that the Version Negotiation packet was not generated by an off-path attacker.

The remainder of the Version Negotiation packet is a list of 32-bit versions which the server supports.

A Version Negotiation packet cannot be explicitly acknowledged in an ACK frame by a client. Receiving another Initial packet implicitly acknowledges a Version Negotiation packet.

The Version Negotiation packet does not include the Packet Number and Length fields present in other packets that use the long header form. Consequently, a Version Negotiation packet consumes an entire UDP datagram.

See Section 6 for a description of the version negotiation process.

17.5. Initial Packet

An Initial packet uses long headers with a type value of 0x7F. It carries the first CRYPTO frames sent by the client and server to perform key exchange, and carries ACKs in either direction.

In order to prevent tampering by version-unaware middleboxes, Initial packets are protected with connection- and version-specific keys (Initial keys) as described in [QUIC-TLS]. This protection does not provide confidentiality or integrity against on-path attackers, but provides some level of protection against off-path attackers.

An Initial packet (shown in Figure 13) has two additional header fields that are added to the Long Header before the Length field.

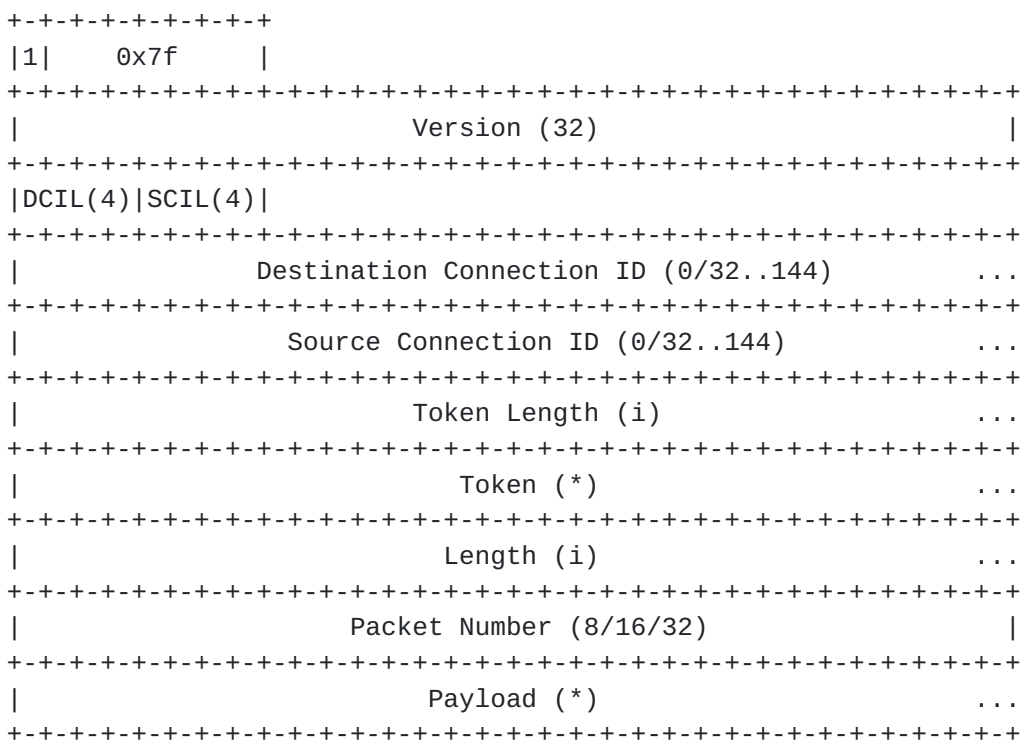


Figure 13: Initial Packet

These fields include the token that was previously provided in a Retry packet or NEW_TOKEN frame:

Token Length: A variable-length integer specifying the length of the Token field, in bytes. This value is zero if no token is present. Initial packets sent by the server MUST set the Token Length field to zero; clients that receive an Initial packet with a non-zero Token Length field MUST either discard the packet or generate a connection error of type PROTOCOL_VIOLATION.

Token: The value of the token.

The client and server use the Initial packet type for any packet that contains an initial cryptographic handshake message. This includes all cases where a new packet containing the initial cryptographic message needs to be created, such as the packets sent after receiving a Version Negotiation (Section 17.4) or Retry packet (Section 17.7).

A server sends its first Initial packet in response to a client Initial. A server may send multiple Initial packets. The cryptographic key exchange could require multiple round trips or retransmissions of this data.

The payload of an Initial packet includes a CRYPTO frame (or frames) containing a cryptographic handshake message, ACK frames, or both. PADDING and CONNECTION_CLOSE frames are also permitted. An endpoint that receives an Initial packet containing other frames can either discard the packet as spurious or treat it as a connection error.

The first packet sent by a client always includes a CRYPTO frame that contains the entirety of the first cryptographic handshake message. This packet, and the cryptographic handshake message, MUST fit in a single UDP datagram (see Section 7). The first CRYPTO frame sent always begins at an offset of 0 (see Section 7).

Note that if the server sends a HelloRetryRequest, the client will send a second Initial packet. This Initial packet will continue the cryptographic handshake and will contain a CRYPTO frame with an offset matching the size of the CRYPTO frame sent in the first Initial packet. Cryptographic handshake messages subsequent to the first do not need to fit within a single UDP datagram.

17.5.1. Starting Packet Numbers

The first Initial packet sent by either endpoint contains a packet number of 0. The packet number MUST increase monotonically thereafter. Initial packets are in a different packet number space to other packets (see Section 12.3).

17.5.2. 0-RTT Packet Numbers

Packet numbers for 0-RTT protected packets use the same space as 1-RTT protected packets.

After a client receives a Retry or Version Negotiation packet, 0-RTT packets are likely to have been lost or discarded by the server. A client MAY attempt to resend data in 0-RTT packets after it sends a new Initial packet.

A client MUST NOT reset the packet number it uses for 0-RTT packets. The keys used to protect 0-RTT packets will not change as a result of responding to a Retry or Version Negotiation packet unless the client also regenerates the cryptographic handshake message. Sending packets with the same packet number in that case is likely to compromise the packet protection for all 0-RTT packets because the same key and nonce could be used to protect different content.

Receiving a Retry or Version Negotiation packet, especially a Retry that changes the connection ID used for subsequent packets, indicates a strong possibility that 0-RTT packets could be lost. A client only receives acknowledgments for its 0-RTT packets once the handshake is

complete. Consequently, a server might expect 0-RTT packets to start with a packet number of 0. Therefore, in determining the length of the packet number encoding for 0-RTT packets, a client MUST assume that all packets up to the current packet number are in flight, starting from a packet number of 0. Thus, 0-RTT packets could need to use a longer packet number encoding.

A client SHOULD instead generate a fresh cryptographic handshake message and start packet numbers from 0. This ensures that new 0-RTT packets will not use the same keys, avoiding any risk of key and nonce reuse; this also prevents 0-RTT packets from previous handshake attempts from being accepted as part of the connection.

17.6. Handshake Packet

A Handshake packet uses long headers with a type value of 0x7D. It is used to carry acknowledgments and cryptographic handshake messages from the server and client.

Once a client has received a Handshake packet from a server, it uses Handshake packets to send subsequent cryptographic handshake messages and acknowledgments to the server.

The Destination Connection ID field in a Handshake packet contains a connection ID that is chosen by the recipient of the packet; the Source Connection ID includes the connection ID that the sender of the packet wishes to use (see Section 7.2).

The first Handshake packet sent by a server contains a packet number of 0. Handshake packets are their own packet number space. Packet numbers are incremented normally for other Handshake packets.

The payload of this packet contains CRYPTO frames and could contain PADDING, or ACK frames. Handshake packets MAY contain CONNECTION_CLOSE or APPLICATION_CLOSE frames. Endpoints MUST treat receipt of Handshake packets with other frames as a connection error.

17.7. Retry Packet

A Retry packet uses a long packet header with a type value of 0x7E. It carries an address validation token created by the server. It is used by a server that wishes to perform a stateless retry (see Section 8.1).

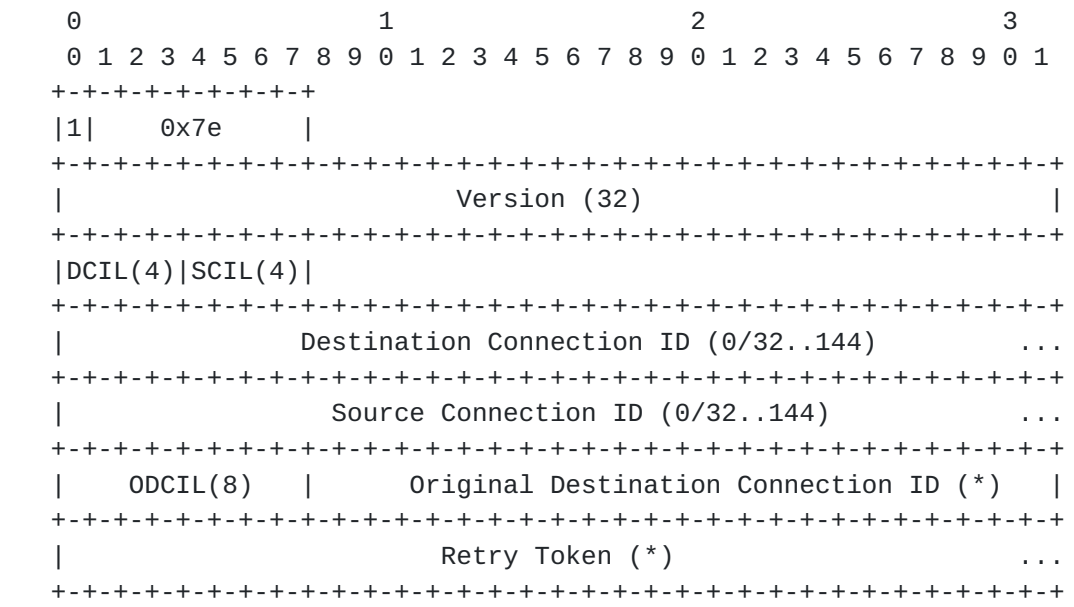


Figure 14: Retry Packet

A Retry packet (shown in Figure 14) only uses the invariant portion of the long packet header [QUIC-INVARIANTS]; that is, the fields up to and including the Destination and Source Connection ID fields. A Retry packet does not contain any protected fields. Like Version Negotiation, a Retry packet contains the long header including the connection IDs, but omits the Length, Packet Number, and Payload fields. These are replaced with:

ODCIL: The length of the Original Destination Connection ID field. The length is encoded in the least significant 4 bits of the octet, using the same encoding as the DCIL and SCIL fields. The most significant 4 bits of this octet are reserved. Unless a use for these bits has been negotiated, endpoints SHOULD send randomized values and MUST ignore any value that it receives.

Original Destination Connection ID: The Original Destination Connection ID contains the value of the Destination Connection ID from the Initial packet that this Retry is in response to. The length of this field is given in ODCIL.

Retry Token: An opaque token that the server can use to validate the client's address.

The server populates the Destination Connection ID with the connection ID that the client included in the Source Connection ID of the Initial packet.

The server includes a connection ID of its choice in the Source Connection ID field. This value MUST not be equal to the Destination Connection ID field of the packet sent by the client. The client MUST use this connection ID in the Destination Connection ID of subsequent packets that it sends.

A server MAY send Retry packets in response to Initial and 0-RTT packets. A server can either discard or buffer 0-RTT packets that it receives. A server can send multiple Retry packets as it receives Initial or 0-RTT packets.

A client MUST accept and process at most one Retry packet for each connection attempt. After the client has received and processed an Initial or Retry packet from the server, it MUST discard any subsequent Retry packets that it receives.

Clients MUST discard Retry packets that contain an Original Destination Connection ID field that does not match the Destination Connection ID from its Initial packet. This prevents an off-path attacker from injecting a Retry packet.

The client responds to a Retry packet with an Initial packet that includes the provided Retry Token to continue connection establishment.

A client sets the Destination Connection ID field of this Initial packet to the value from the Source Connection ID in the Retry packet. Changing Destination Connection ID also results in a change to the keys used to protect the Initial packet. It also sets the Token field to the token provided in the Retry. The client MUST NOT change the Source Connection ID because the server could include the connection ID as part of its token validation logic (see Section 8.1.2).

All subsequent Initial packets from the client MUST use the connection ID and token values from the Retry packet. Aside from this, the Initial packet sent by the client is subject to the same restrictions as the first Initial packet. A client can either reuse the cryptographic handshake message or construct a new one at its discretion.

A client MAY attempt 0-RTT after receiving a Retry packet by sending 0-RTT packets to the connection ID provided by the server. A client that sends additional 0-RTT packets without constructing a new cryptographic handshake message MUST NOT reset the packet number to 0 after a Retry packet, see Section 17.5.2.

A server acknowledges the use of a Retry packet for a connection using the original_connection_id transport parameter (see Section 18.1). If the server sends a Retry packet, it MUST include the value of the Original Destination Connection ID field of the Retry packet (that is, the Destination Connection ID field from the client's first Initial packet) in the transport parameter.

If the client received and processed a Retry packet, it validates that the original_connection_id transport parameter is present and correct; otherwise, it validates that the transport parameter is absent. A client MUST treat a failed validation as a connection error of type TRANSPORT_PARAMETER_ERROR.

A Retry packet does not include a packet number and cannot be explicitly acknowledged by a client.

18. Transport Parameter Encoding

The format of the transport parameters is the TransportParameters struct from Figure 15. This is described using the presentation language from Section 3 of [TLS13].

```
uint32 QuicVersion;

enum {
   initial_max_stream_data_bidi_local(0),
   initial_max_data(1),
   initial_max_bidi_streams(2),
   idle_timeout(3),
   preferred_address(4),
   max_packet_size(5),
   stateless_reset_token(6),
   ack_delay_exponent(7),
   initial_max_uni_streams(8),
   disable_migration(9),
   initial_max_stream_data_bidi_remote(10),
   initial_max_stream_data_uni(11),
   max_ack_delay(12),
   original_connection_id(13),
   (65535)
} TransportParameterId;

struct {
   TransportParameterId parameter;
   opaque value<0..2^16-1>;
} TransportParameter;

struct {
   select (Handshake.msg_type) {
      case client_hello:
         QuicVersion initial_version;

      case encrypted_extensions:
         QuicVersion negotiated_version;
         QuicVersion supported_versions<4..2^8-4>;
   };
   TransportParameter parameters<0..2^16-1>;
} TransportParameters;

struct {
  enum { IPv4(4), IPv6(6), (15) } ipVersion;
  opaque ipAddress<4..2^8-1>;
  uint16 port;
  opaque connectionId<0..18>;
  opaque statelessResetToken[16];
} PreferredAddress;
```

Figure 15: Definition of TransportParameters

The "extension_data" field of the quic_transport_parameters extension defined in [QUIC-TLS] contains a TransportParameters value. TLS encoding rules are therefore used to describe the encoding of transport parameters.

QUIC encodes transport parameters into a sequence of octets, which are then included in the cryptographic handshake.

18.1. Transport Parameter Definitions

An endpoint MAY use the following transport parameters:

idle_timeout (0x0003): The idle timeout is a value in seconds that is encoded as an unsigned 16-bit integer. If this parameter is absent or zero then the idle timeout is disabled.

max_packet_size (0x0005): The maximum packet size parameter places a limit on the size of packets that the endpoint is willing to receive, encoded as an unsigned 16-bit integer. This indicates that packets larger than this limit will be dropped. The default for this parameter is the maximum permitted UDP payload of 65527. Values below 1200 are invalid. This limit only applies to protected packets (Section 12.1).

ack_delay_exponent (0x0007): An 8-bit unsigned integer value indicating an exponent used to decode the ACK Delay field in the ACK frame, see Section 19.15. If this value is absent, a default value of 3 is assumed (indicating a multiplier of 8). The default value is also used for ACK frames that are sent in Initial and Handshake packets. Values above 20 are invalid.

disable_migration (0x0009): The endpoint does not support connection migration (Section 9). Peers MUST NOT send any packets, including probing packets (Section 9.1), from a local address other than that used to perform the handshake. This parameter is a zero-length value.

max_ack_delay (0x000c): An 8 bit unsigned integer value indicating the maximum amount of time in milliseconds by which the endpoint will delay sending acknowledgments. If this value is absent, a default of 25 milliseconds is assumed.

Either peer MAY advertise an initial value for flow control of each type of stream on which they might receive data. Each of the following transport parameters is encoded as an unsigned 32-bit integer in units of octets:

initial_max_stream_data_bidi_local (0x0000): The initial stream maximum data for bidirectional, locally-initiated streams parameter contains the initial flow control limit for newly created bidirectional streams opened by the endpoint that sets the transport parameter. In client transport parameters, this applies to streams with an identifier ending in 0x0; in server transport parameters, this applies to streams ending in 0x1.

initial_max_stream_data_bidi_remote (0x000a): The initial stream maximum data for bidirectional, peer-initiated streams parameter contains the initial flow control limit for newly created bidirectional streams opened by the endpoint that receives the transport parameter. In client transport parameters, this applies to streams with an identifier ending in 0x1; in server transport parameters, this applies to streams ending in 0x0.

initial_max_stream_data_uni (0x000b): The initial stream maximum data for unidirectional streams parameter contains the initial flow control limit for newly created unidirectional streams opened by the endpoint that receives the transport parameter. In client transport parameters, this applies to streams with an identifier ending in 0x3; in server transport parameters, this applies to streams ending in 0x2.

If present, transport parameters that set initial flow control limits (initial_max_stream_data_bidi_local, initial_max_stream_data_bidi_remote, and initial_max_stream_data_uni) are equivalent to sending a MAX_STREAM_DATA frame (Section 19.6) on every stream of the corresponding type immediately after opening. If the transport parameter is absent, streams of that type start with a flow control limit of 0.

initial_max_data (0x0001): The initial maximum data parameter contains the initial value for the maximum amount of data that can be sent on the connection. This parameter is encoded as an unsigned 32-bit integer in units of octets. This is equivalent to sending a MAX_DATA (Section 19.5) for the connection immediately after completing the handshake. If the transport parameter is absent, the connection starts with a flow control limit of 0.

initial_max_bidi_streams (0x0002): The initial maximum bidirectional streams parameter contains the initial maximum number of bidirectional streams the peer may initiate, encoded as an unsigned 16-bit integer. If this parameter is absent or zero, bidirectional streams cannot be created until a MAX_STREAM_ID frame is sent. Setting this parameter is equivalent to sending a MAX_STREAM_ID (Section 19.7) immediately after completing the handshake containing the corresponding Stream ID. For example, a

value of 0x05 would be equivalent to receiving a MAX_STREAM_ID containing 16 when received by a client or 17 when received by a server.

initial_max_uni_streams (0x0008): The initial maximum unidirectional streams parameter contains the initial maximum number of unidirectional streams the peer may initiate, encoded as an unsigned 16-bit integer. If this parameter is absent or zero, unidirectional streams cannot be created until a MAX_STREAM_ID frame is sent. Setting this parameter is equivalent to sending a MAX_STREAM_ID (Section 19.7) immediately after completing the handshake containing the corresponding Stream ID. For example, a value of 0x05 would be equivalent to receiving a MAX_STREAM_ID containing 18 when received by a client or 19 when received by a server.

A server MUST include the following transport parameter if it sent a Retry packet:

original_connection_id (0x000d): The value of the Destination Connection ID field from the first Initial packet sent by the client. This transport parameter is only sent by the server.

A server MAY include the following transport parameters:

stateless_reset_token (0x0006): The Stateless Reset Token is used in verifying a stateless reset, see Section 10.4. This parameter is a sequence of 16 octets.

preferred_address (0x0004): The server's Preferred Address is used to effect a change in server address at the end of the handshake, as described in Section 9.6.

A client MUST NOT include an original connection ID, a stateless reset token, or a preferred address. A server MUST treat receipt of any of these transport parameters as a connection error of type TRANSPORT_PARAMETER_ERROR.

19. Frame Types and Formats

As described in Section 12.4, packets contain one or more frames. This section describes the format and semantics of the core QUIC frame types.

19.1. PADDING Frame

The PADDING frame (type=0x00) has no semantic value. PADDING frames can be used to increase the size of a packet. Padding can be used to increase an initial client packet to the minimum required size, or to provide protection against traffic analysis for protected packets.

A PADDING frame has no content. That is, a PADDING frame consists of the single octet that identifies the frame as a PADDING frame.

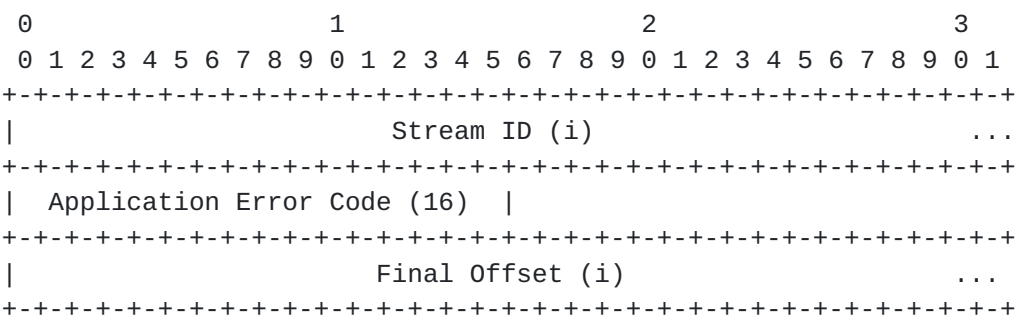
19.2. RST_STREAM Frame

An endpoint may use a RST_STREAM frame (type=0x01) to abruptly terminate a stream.

After sending a RST_STREAM, an endpoint ceases transmission and retransmission of STREAM frames on the identified stream. A receiver of RST_STREAM can discard any data that it already received on that stream.

An endpoint that receives a RST_STREAM frame for a send-only stream MUST terminate the connection with error PROTOCOL_VIOLATION.

The RST_STREAM frame is as follows:



The fields are:

Stream ID: A variable-length integer encoding of the Stream ID of the stream being terminated.

Application Protocol Error Code: A 16-bit application protocol error code (see Section 20.1) which indicates why the stream is being closed.

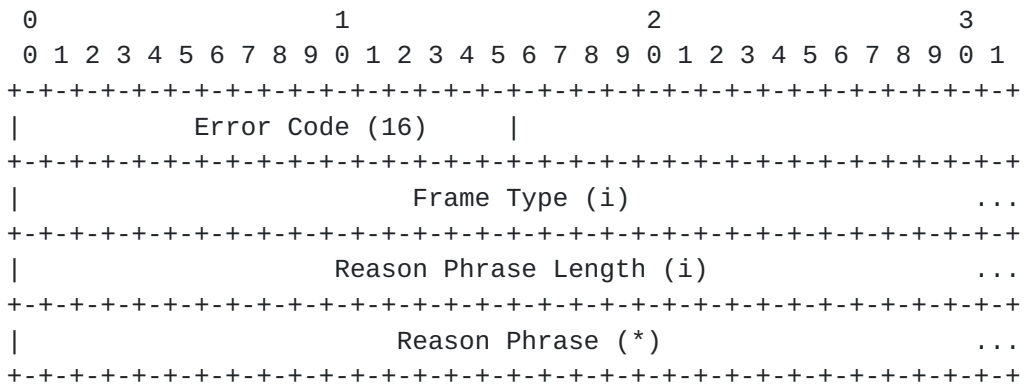
Final Offset: A variable-length integer indicating the absolute byte offset of the end of data written on this stream by the RST_STREAM sender.

19.3. CONNECTION_CLOSE frame

An endpoint sends a CONNECTION_CLOSE frame (type=0x02) to notify its peer that the connection is being closed. CONNECTION_CLOSE is used to signal errors at the QUIC layer, or the absence of errors (with the NO_ERROR code).

If there are open streams that haven't been explicitly closed, they are implicitly closed when the connection is closed.

The CONNECTION_CLOSE frame is as follows:



The fields of a CONNECTION_CLOSE frame are as follows:

Error Code: A 16-bit error code which indicates the reason for closing this connection. CONNECTION_CLOSE uses codes from the space defined in Section 20.

Frame Type: A variable-length integer encoding the type of frame that triggered the error. A value of 0 (equivalent to the mention of the PADDING frame) is used when the frame type is unknown.

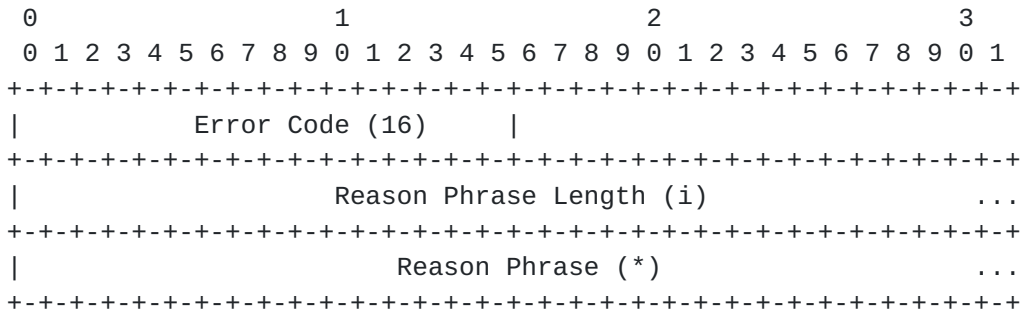
Reason Phrase Length: A variable-length integer specifying the length of the reason phrase in bytes. Note that a CONNECTION_CLOSE frame cannot be split between packets, so in practice any limits on packet size will also limit the space available for a reason phrase.

Reason Phrase: A human-readable explanation for why the connection was closed. This can be zero length if the sender chooses to not give details beyond the Error Code. This SHOULD be a UTF-8 encoded string [RFC3629].

19.4. APPLICATION_CLOSE frame

An APPLICATION_CLOSE frame (type=0x03) is used to signal an error with the protocol that uses QUIC.

The APPLICATION_CLOSE frame is as follows:



The fields of an APPLICATION_CLOSE frame are as follows:

Error Code: A 16-bit error code which indicates the reason for closing this connection. APPLICATION_CLOSE uses codes from the application protocol error code space, see Section 20.1.

Reason Phrase Length: This field is identical in format and semantics to the Reason Phrase Length field from CONNECTION_CLOSE.

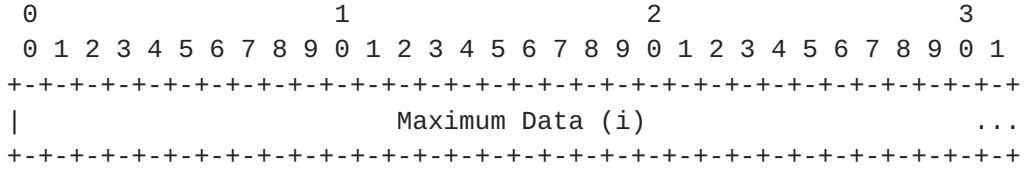
Reason Phrase: This field is identical in format and semantics to the Reason Phrase field from CONNECTION_CLOSE.

APPLICATION_CLOSE has similar format and semantics to the CONNECTION_CLOSE frame (Section 19.3). Aside from the semantics of the Error Code field and the omission of the Frame Type field, both frames are used to close the connection.

19.5. MAX_DATA Frame

The MAX_DATA frame (type=0x04) is used in flow control to inform the peer of the maximum amount of data that can be sent on the connection as a whole.

The frame is as follows:



The fields in the MAX_DATA frame are as follows:

Maximum Data: A variable-length integer indicating the maximum amount of data that can be sent on the entire connection, in units of octets.

All data sent in STREAM frames counts toward this limit. The sum of the largest received offsets on all streams - including streams in terminal states - MUST NOT exceed the value advertised by a receiver. An endpoint MUST terminate a connection with a FLOW_CONTROL_ERROR error if it receives more data than the maximum data value that it has sent, unless this is a result of a change in the initial limits (see Section 7.3.1).

19.6. MAX_STREAM_DATA Frame

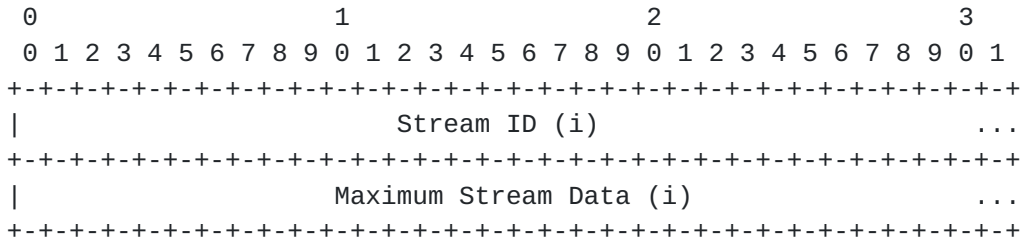
The MAX_STREAM_DATA frame (type=0x05) is used in flow control to inform a peer of the maximum amount of data that can be sent on a stream.

An endpoint that receives a MAX_STREAM_DATA frame for a receive-only stream MUST terminate the connection with error PROTOCOL_VIOLATION.

An endpoint that receives a MAX_STREAM_DATA frame for a send-only stream it has not opened MUST terminate the connection with error PROTOCOL_VIOLATION.

Note that an endpoint may legally receive a MAX_STREAM_DATA frame on a bidirectional stream it has not opened.

The frame is as follows:



The fields in the MAX_STREAM_DATA frame are as follows:

Stream ID: The stream ID of the stream that is affected encoded as a variable-length integer.

Maximum Stream Data: A variable-length integer indicating the maximum amount of data that can be sent on the identified stream, in units of octets.

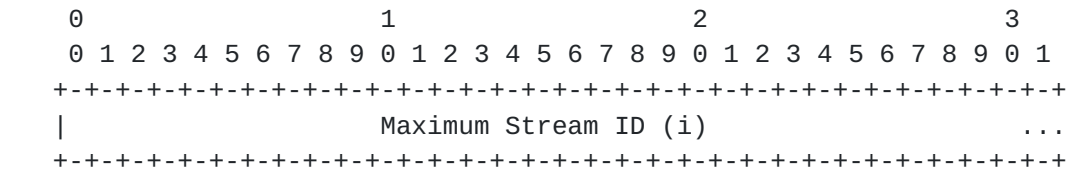
When counting data toward this limit, an endpoint accounts for the largest received offset of data that is sent or received on the stream. Loss or reordering can mean that the largest received offset on a stream can be greater than the total size of data received on that stream. Receiving STREAM frames might not increase the largest received offset.

The data sent on a stream MUST NOT exceed the largest maximum stream data value advertised by the receiver. An endpoint MUST terminate a connection with a FLOW_CONTROL_ERROR error if it receives more data than the largest maximum stream data that it has sent for the affected stream, unless this is a result of a change in the initial limits (see Section 7.3.1).

19.7. MAX_STREAM_ID Frame

The MAX_STREAM_ID frame (type=0x06) informs the peer of the maximum stream ID that they are permitted to open.

The frame is as follows:



The fields in the MAX_STREAM_ID frame are as follows:

Maximum Stream ID: ID of the maximum unidirectional or bidirectional peer-initiated stream ID for the connection encoded as a variable-length integer. The limit applies to unidirectional steams if the second least signification bit of the stream ID is 1, and applies to bidirectional streams if it is 0.

Loss or reordering can mean that a MAX_STREAM_ID frame can be received which states a lower stream limit than the client has previously received. MAX_STREAM_ID frames which do not increase the maximum stream ID MUST be ignored.

A peer MUST NOT initiate a stream with a higher stream ID than the greatest maximum stream ID it has received. An endpoint MUST terminate a connection with a STREAM_ID_ERROR error if a peer initiates a stream with a higher stream ID than it has sent, unless this is a result of a change in the initial limits (see Section 7.3.1).

19.8. PING Frame

Endpoints can use PING frames (type=0x07) to verify that their peers are still alive or to check reachability to the peer. The PING frame contains no additional fields.

The receiver of a PING frame simply needs to acknowledge the packet containing this frame.

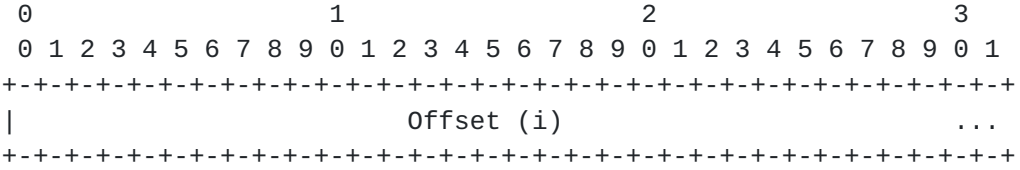
The PING frame can be used to keep a connection alive when an application or application protocol wishes to prevent the connection from timing out. An application protocol SHOULD provide guidance about the conditions under which generating a PING is recommended. This guidance SHOULD indicate whether it is the client or the server that is expected to send the PING. Having both endpoints send PING frames without coordination can produce an excessive number of packets and poor performance.

A connection will time out if no packets are sent or received for a period longer than the time specified in the idle_timeout transport parameter (see Section 10). However, state in middleboxes might time out earlier than that. Though REQ-5 in [RFC4787] recommends a 2 minute timeout interval, experience shows that sending packets every 15 to 30 seconds is necessary to prevent the majority of middleboxes from losing state for UDP flows.

19.9. BLOCKED Frame

A sender SHOULD send a BLOCKED frame (type=0x08) when it wishes to send data, but is unable to due to connection-level flow control (see Section 4). BLOCKED frames can be used as input to tuning of flow control algorithms (see Section 4.2).

The BLOCKED frame is as follows:



The BLOCKED frame contains a single field.

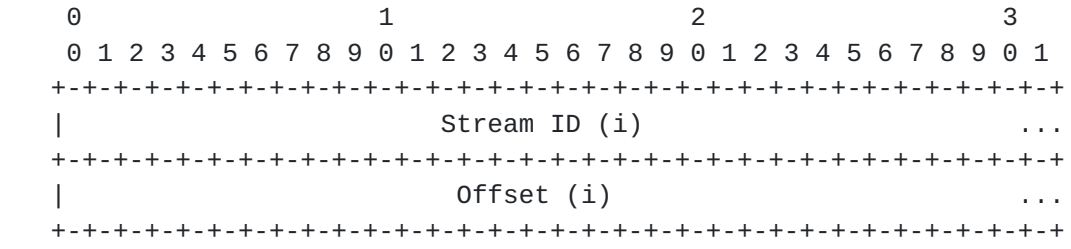
Offset: A variable-length integer indicating the connection-level offset at which the blocking occurred.

19.10. STREAM_BLOCKED Frame

A sender SHOULD send a STREAM_BLOCKED frame (type=0x09) when it wishes to send data, but is unable to due to stream-level flow control. This frame is analogous to BLOCKED (Section 19.9).

An endpoint that receives a STREAM_BLOCKED frame for a send-only stream MUST terminate the connection with error PROTOCOL_VIOLATION.

The STREAM_BLOCKED frame is as follows:



The STREAM_BLOCKED frame contains two fields:

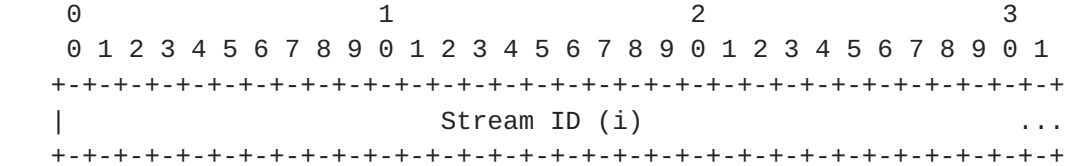
Stream ID: A variable-length integer indicating the stream which is flow control blocked.

Offset: A variable-length integer indicating the offset of the stream at which the blocking occurred.

19.11. STREAM_ID_BLOCKED Frame

A sender SHOULD send a STREAM_ID_BLOCKED frame (type=0x0a) when it wishes to open a stream, but is unable to due to the maximum stream ID limit set by its peer (see Section 19.7). This does not open the stream, but informs the peer that a new stream was needed, but the stream limit prevented the creation of the stream.

The STREAM_ID_BLOCKED frame is as follows:



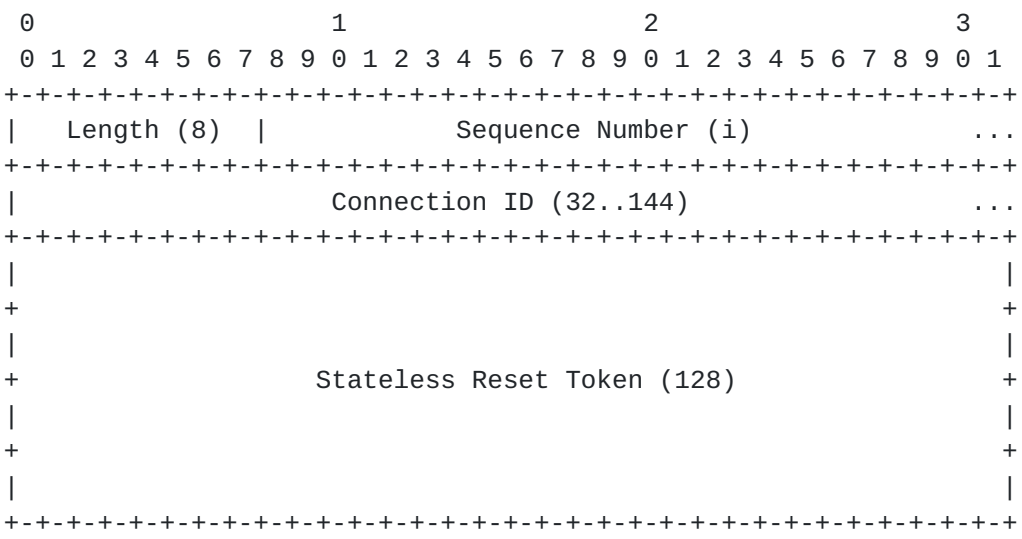
The STREAM_ID_BLOCKED frame contains a single field.

Stream ID: A variable-length integer indicating the highest stream ID that the sender was permitted to open.

19.12. NEW_CONNECTION_ID Frame

An endpoint sends a NEW_CONNECTION_ID frame (type=0x0b) to provide its peer with alternative connection IDs that can be used to break linkability when migrating connections (see Section 9.5).

The NEW_CONNECTION_ID frame is as follows:



The fields are:

Length: An 8-bit unsigned integer containing the length of the connection ID. Values less than 4 and greater than 18 are invalid and MUST be treated as a connection error of type PROTOCOL_VIOLATION.

Sequence Number: The sequence number assigned to the connection ID by the sender. See Section 5.1.1.

Connection ID: A connection ID of the specified length.

Stateless Reset Token: A 128-bit value that will be used for a stateless reset when the associated connection ID is used (see Section 10.4).

An endpoint MUST NOT send this frame if it currently requires that its peer send packets with a zero-length Destination Connection ID. Changing the length of a connection ID to or from zero-length makes it difficult to identify when the value of the connection ID changed. An endpoint that is sending packets with a zero-length Destination Connection ID MUST treat receipt of a NEW_CONNECTION_ID frame as a connection error of type PROTOCOL_VIOLATION.

Transmission errors, timeouts and retransmissions might cause the same NEW_CONNECTION_ID frame to be received multiple times. Receipt of the same frame multiple times MUST NOT be treated as a connection error. A receiver can use the sequence number supplied in the NEW_CONNECTION_ID frame to identify new connection IDs from old ones.

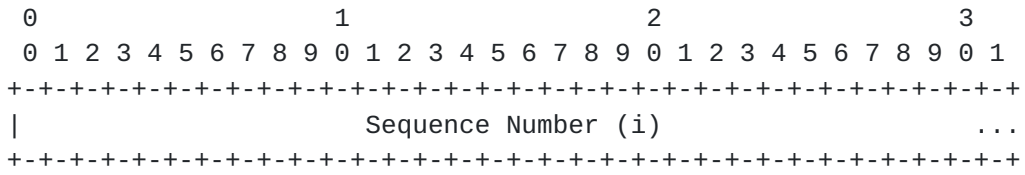
If an endpoint receives a NEW_CONNECTION_ID frame that repeats a previously issued connection ID with a different Stateless Reset Token or a different sequence number, the endpoint MAY treat that receipt as a connection error of type PROTOCOL_VIOLATION.

19.13. RETIRE_CONNECTION_ID Frame

An endpoint sends a RETIRE_CONNECTION_ID frame (type=0x1b) to indicate that it will no longer use a connection ID that was issued by its peer. This may include the connection ID provided during the handshake. Sending a RETIRE_CONNECTION_ID frame also serves as a request to the peer to send additional connection IDs for future use (see Section 5.1). New connection IDs can be delivered to a peer using the NEW_CONNECTION_ID frame (Section 19.12).

Retiring a connection ID invalidates the stateless reset token associated with that connection ID.

The RETIRE_CONNECTION_ID frame is as follows:



The fields are:

Sequence Number: The sequence number of the connection ID being retired. See Section 5.1.2.

Receipt of a RETIRE_CONNECTION_ID frame containing a sequence number greater than any previously sent to the peer MAY be treated as a connection error of type PROTOCOL_VIOLATION.

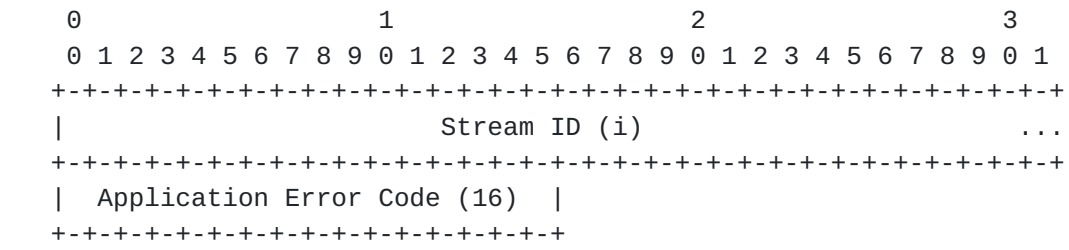
An endpoint cannot send this frame if it was provided with a zero-length connection ID by its peer. An endpoint that provides a zero-length connection ID MUST treat receipt of a RETIRE_CONNECTION_ID frame as a connection error of type PROTOCOL_VIOLATION.

19.14. STOP_SENDING Frame

An endpoint may use a STOP_SENDING frame (type=0x0c) to communicate that incoming data is being discarded on receipt at application request. This signals a peer to abruptly terminate transmission on a stream.

Receipt of a STOP_SENDING frame is only valid for a send stream that exists and is not in the "Ready" state (see Section 3.1). Receiving a STOP_SENDING frame for a send stream that is "Ready" or non-existent MUST be treated as a connection error of type PROTOCOL_VIOLATION. An endpoint that receives a STOP_SENDING frame for a receive-only stream MUST terminate the connection with error PROTOCOL_VIOLATION.

The STOP_SENDING frame is as follows:



The fields are:

Stream ID: A variable-length integer carrying the Stream ID of the stream being ignored.

Application Error Code: A 16-bit, application-specified reason the sender is ignoring the stream (see Section 20.1).

19.15. ACK Frame

Receivers send ACK frames (types 0x1a and 0x1b) to inform senders of packets they have received and processed. The ACK frame contains one or more ACK Blocks. ACK Blocks are ranges of acknowledged packets. If the frame type is 0x1b, ACK frames also contain the sum of ECN marks received on the connection up until this point.

QUIC acknowledgements are irrevocable. Once acknowledged, a packet remains acknowledged, even if it does not appear in a future ACK frame. This is unlike TCP SACKs ([RFC2018]).

It is expected that a sender will reuse the same packet number across different packet number spaces. ACK frames only acknowledge the

packet numbers that were transmitted by the sender in the same packet number space of the packet that the ACK was received in.

Version Negotiation and Retry packets cannot be acknowledged because they do not contain a packet number. Rather than relying on ACK frames, these packets are implicitly acknowledged by the next Initial packet sent by the client.

An ACK frame is shown below.

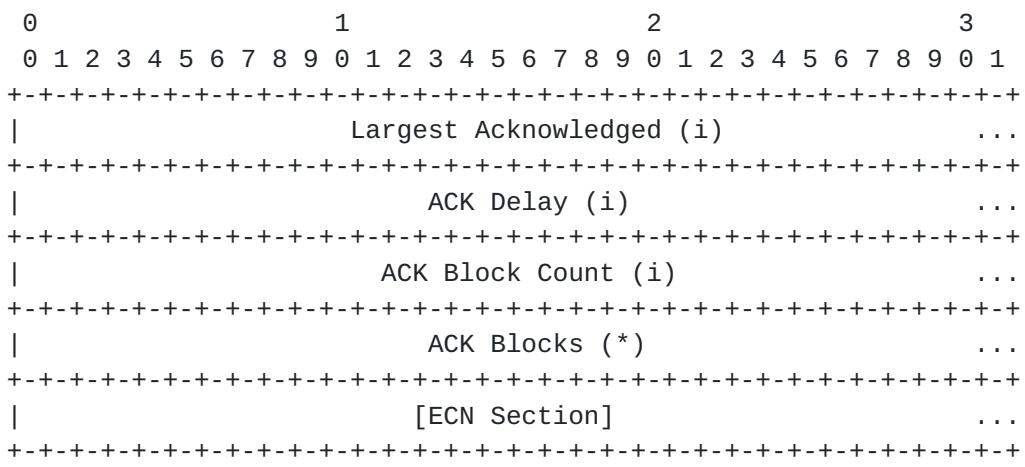


Figure 16: ACK Frame Format

The fields in the ACK frame are as follows:

Largest Acknowledged: A variable-length integer representing the largest packet number the peer is acknowledging; this is usually the largest packet number that the peer has received prior to generating the ACK frame. Unlike the packet number in the QUIC long or short header, the value in an ACK frame is not truncated.

ACK Delay: A variable-length integer including the time in microseconds that the largest acknowledged packet, as indicated in the Largest Acknowledged field, was received by this peer to when this ACK was sent. The value of the ACK Delay field is scaled by multiplying the encoded value by 2 to the power of the value of the "ack_delay_exponent" transport parameter set by the sender of the ACK frame. The "ack_delay_exponent" defaults to 3, or a multiplier of 8 (see Section 18.1). Scaling in this fashion allows for a larger range of values with a shorter encoding at the cost of lower resolution.

ACK Block Count: A variable-length integer specifying the number of Additional ACK Block (and Gap) fields after the First ACK Block.

ACK Blocks: Contains one or more blocks of packet numbers which have been successfully received, see Section 19.15.1.

19.15.1. ACK Block Section

The ACK Block Section consists of alternating Gap and ACK Block fields in descending packet number order. A First Ack Block field is followed by a variable number of alternating Gap and Additional ACK Blocks. The number of Gap and Additional ACK Block fields is determined by the ACK Block Count field.

Gap and ACK Block fields use a relative integer encoding for efficiency. Though each encoded value is positive, the values are subtracted, so that each ACK Block describes progressively lower-numbered packets. As long as contiguous ranges of packets are small, the variable-length integer encoding ensures that each range can be expressed in a small number of octets.

The ACK frame uses the least significant bit(bit (that is, type 0x1b) to indicate ECN feedback and report receipt of packets with ECN codepoints of ECT(0), ECT(1), or CE in the packet's IP header.

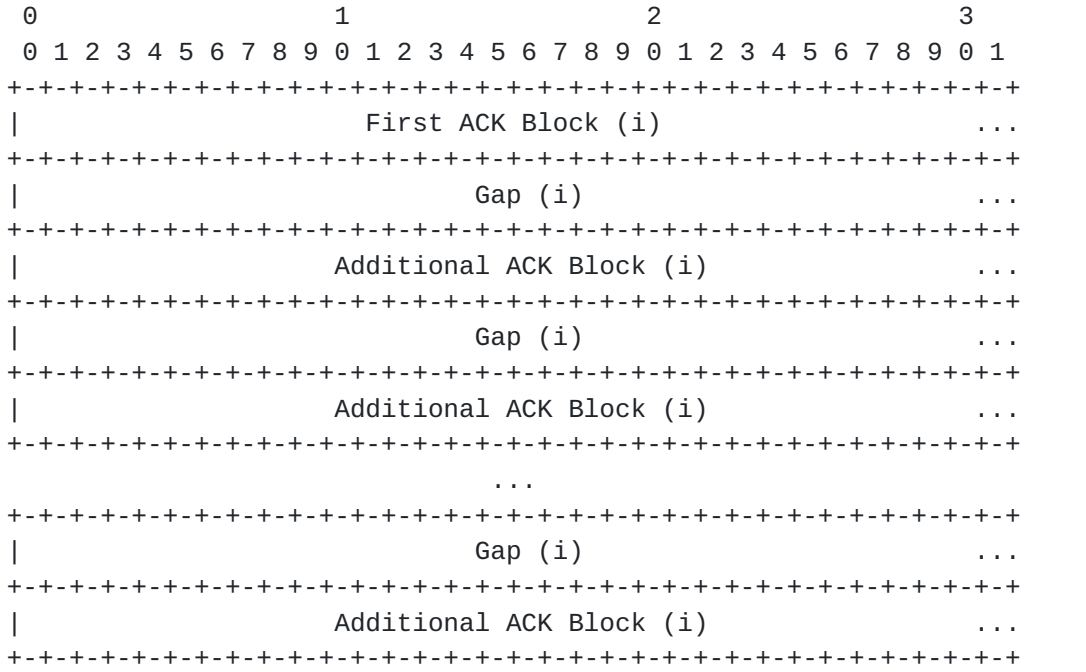


Figure 17: ACK Block Section

Each ACK Block acknowledges a contiguous range of packets by indicating the number of acknowledged packets that precede the largest packet number in that block. A value of zero indicates that only the largest packet number is acknowledged. Larger ACK Block

values indicate a larger range, with corresponding lower values for the smallest packet number in the range. Thus, given a largest packet number for the ACK, the smallest value is determined by the formula:

```
smallest = largest - ack_block
```

The range of packets that are acknowledged by the ACK Block include the range from the smallest packet number to the largest, inclusive.

The largest value for the First ACK Block is determined by the Largest Acknowledged field; the largest for Additional ACK Blocks is determined by cumulatively subtracting the size of all preceding ACK Blocks and Gaps.

Each Gap indicates a range of packets that are not being acknowledged. The number of packets in the gap is one higher than the encoded value of the Gap Field.

The value of the Gap field establishes the largest packet number value for the ACK Block that follows the gap using the following formula:

```
largest = previous_smallest - gap - 2
```

If the calculated value for largest or smallest packet number for any ACK Block is negative, an endpoint MUST generate a connection error of type FRAME_ENCODING_ERROR indicating an error in an ACK frame.

The fields in the ACK Block Section are:

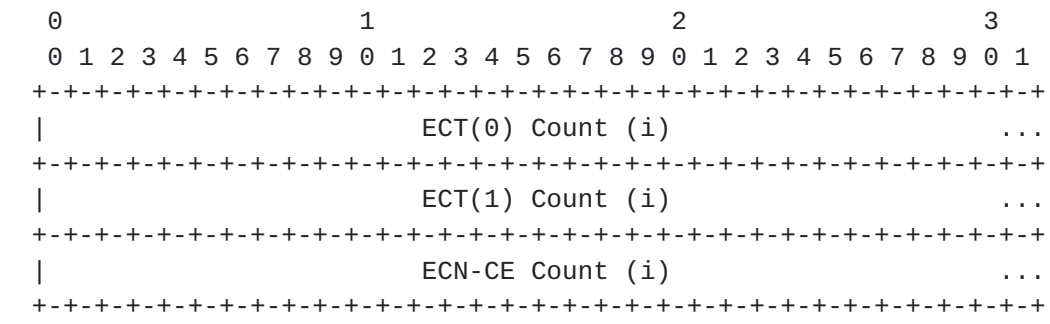
First ACK Block: A variable-length integer indicating the number of contiguous packets preceding the Largest Acknowledged that are being acknowledged.

Gap (repeated): A variable-length integer indicating the number of contiguous unacknowledged packets preceding the packet number one lower than the smallest in the preceding ACK Block.

Additional ACK Block (repeated): A variable-length integer indicating the number of contiguous acknowledged packets preceding the largest packet number, as determined by the preceding Gap.

19.15.2. ECN section

The ECN section should only be parsed when the ACK frame type byte is 0x1b. The ECN section consists of 3 ECN counters as shown below.



ECT(0) Count: A variable-length integer representing the total number packets received with the ECT(0) codepoint.

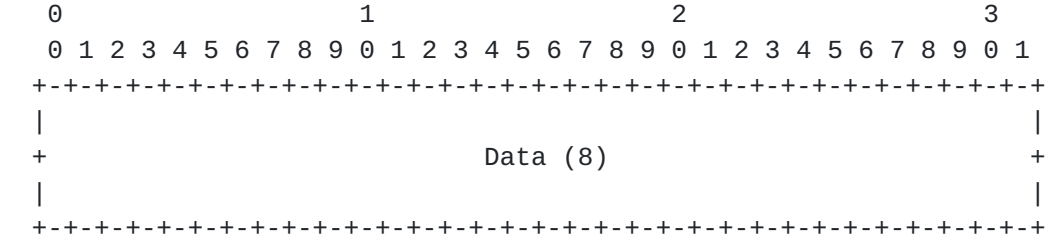
ECT(1) Count: A variable-length integer representing the total number packets received with the ECT(1) codepoint.

CE Count: A variable-length integer representing the total number packets received with the CE codepoint.

19.16. PATH_CHALLENGE Frame

Endpoints can use PATH_CHALLENGE frames (type=0x0e) to check reachability to the peer and for path validation during connection migration.

PATH_CHALLENGE frames contain an 8-byte payload.



Data: This 8-byte field contains arbitrary data.

A PATH_CHALLENGE frame containing 8 octets that are hard to guess is sufficient to ensure that it is easier to receive the packet than it is to guess the value correctly.

The recipient of this frame MUST generate a PATH_RESPONSE frame (Section 19.17) containing the same Data.

19.17. PATH_RESPONSE Frame

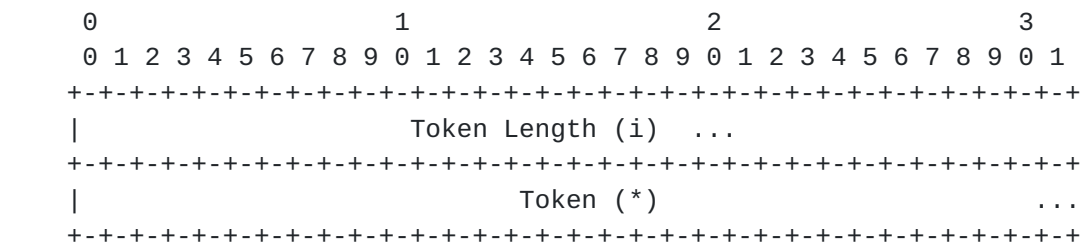
The PATH_RESPONSE frame (type=0x0f) is sent in response to a PATH_CHALLENGE frame. Its format is identical to the PATH_CHALLENGE frame (Section 19.16).

If the content of a PATH_RESPONSE frame does not match the content of a PATH_CHALLENGE frame previously sent by the endpoint, the endpoint MAY generate a connection error of type PROTOCOL_VIOLATION.

19.18. NEW_TOKEN frame

A server sends a NEW_TOKEN frame (type=0x19) to provide the client a token to send in the header of an Initial packet for a future connection.

The NEW_TOKEN frame is as follows:



The fields of a NEW_TOKEN frame are as follows:

Token Length: A variable-length integer specifying the length of the token in bytes.

Token: An opaque blob that the client may use with a future Initial packet.

19.19. STREAM Frames

STREAM frames implicitly create a stream and carry stream data. The STREAM frame takes the form 0b00010XXX (or the set of values from 0x10 to 0x17). The value of the three low-order bits of the frame type determine the fields that are present in the frame.

- The OFF bit (0x04) in the frame type is set to indicate that there is an Offset field present. When set to 1, the Offset field is present; when set to 0, the Offset field is absent and the Stream Data starts at an offset of 0 (that is, the frame contains the first octets of the stream, or the end of a stream that includes no data).

- The LEN bit (0x02) in the frame type is set to indicate that there is a Length field present. If this bit is set to 0, the Length field is absent and the Stream Data field extends to the end of the packet. If this bit is set to 1, the Length field is present.

- The FIN bit (0x01) of the frame type is set only on frames that contain the final offset of the stream. Setting this bit indicates that the frame marks the end of the stream.

An endpoint that receives a STREAM frame for a send-only stream MUST terminate the connection with error PROTOCOL_VIOLATION.

A STREAM frame is shown below.

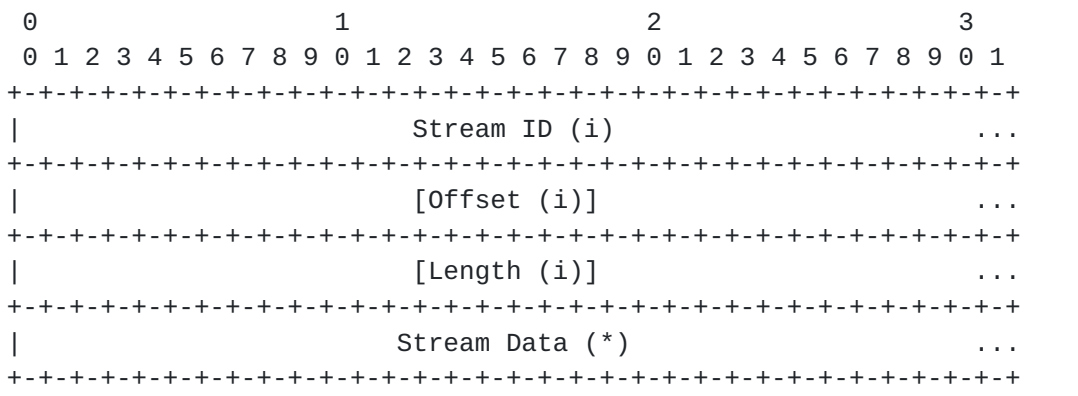


Figure 18: STREAM Frame Format

The STREAM frame contains the following fields:

Stream ID: A variable-length integer indicating the stream ID of the stream (see Section 2.1).

Offset: A variable-length integer specifying the byte offset in the stream for the data in this STREAM frame. This field is present when the OFF bit is set to 1. When the Offset field is absent, the offset is 0.

Length: A variable-length integer specifying the length of the Stream Data field in this STREAM frame. This field is present when the LEN bit is set to 1. When the LEN bit is set to 0, the Stream Data field consumes all the remaining octets in the packet.

Stream Data: The bytes from the designated stream to be delivered.

When a Stream Data field has a length of 0, the offset in the STREAM frame is the offset of the next byte that would be sent.

The first byte in the stream has an offset of 0. The largest offset delivered on a stream - the sum of the re-constructed offset and data length - MUST be less than 2^62.

19.20. CRYPTO Frame

The CRYPTO frame (type=0x18) is used to transmit cryptographic handshake messages. It can be sent in all packet types. The CRYPTO frame offers the cryptographic protocol an in-order stream of bytes. CRYPTO frames are functionally identical to STREAM frames, except that they do not bear a stream identifier; they are not flow controlled; and they do not carry markers for optional offset, optional length, and the end of the stream.

A CRYPTO frame is shown below.

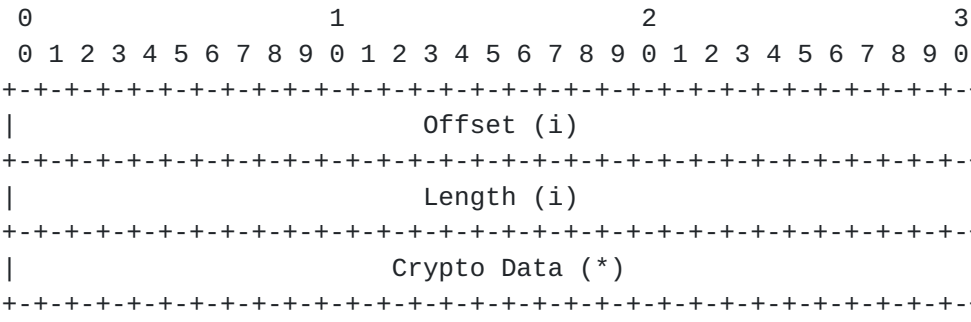


Figure 19: CRYPTO Frame Format

The CRYPTO frame contains the following fields:

Offset: A variable-length integer specifying the byte offset in the stream for the data in this CRYPTO frame.

Length: A variable-length integer specifying the length of the Crypto Data field in this CRYPTO frame.

Crypto Data: The cryptographic message data.

There is a separate flow of cryptographic handshake data in each encryption level, each of which starts at an offset of 0. This implies that each encryption level is treated as a separate CRYPTO stream of data.

Unlike STREAM frames, which include a Stream ID indicating to which stream the data belongs, the CRYPTO frame carries data for a single stream per encryption level. The stream does not have an explicit end, so CRYPTO frames do not have a FIN bit.

19.21. Extension Frames

QUIC frames do not use a self-describing encoding. An endpoint therefore needs to understand the syntax of all frames before it can successfully process a packet. This allows for efficient encoding of frames, but it means that an endpoint cannot send a frame of a type that is unknown to its peer.

An extension to QUIC that wishes to use a new type of frame MUST first ensure that a peer is able to understand the frame. An endpoint can use a transport parameter to signal its willingness to receive one or more extension frame types with the one transport parameter.

Extension frames MUST be congestion controlled and MUST cause an ACK frame to be sent. The exception is extension frames that replace or supplement the ACK frame. Extension frames are not included in flow control unless specified in the extension.

An IANA registry is used to manage the assignment of frame types, see Section 22.2.

20. Transport Error Codes

QUIC error codes are 16-bit unsigned integers.

This section lists the defined QUIC transport error codes that may be used in a CONNECTION_CLOSE frame. These errors apply to the entire connection.

NO_ERROR (0x0): An endpoint uses this with CONNECTION_CLOSE to signal that the connection is being closed abruptly in the absence of any error.

INTERNAL_ERROR (0x1): The endpoint encountered an internal error and cannot continue with the connection.

SERVER_BUSY (0x2): The server is currently busy and does not accept any new connections.

FLOW_CONTROL_ERROR (0x3): An endpoint received more data than it permitted in its advertised data limits (see Section 4).

STREAM_ID_ERROR (0x4): An endpoint received a frame for a stream identifier that exceeded its advertised maximum stream ID.

STREAM_STATE_ERROR (0x5): An endpoint received a frame for a stream that was not in a state that permitted that frame (see Section 3).

FINAL_OFFSET_ERROR (0x6): An endpoint received a STREAM frame containing data that exceeded the previously established final offset. Or an endpoint received a RST_STREAM frame containing a final offset that was lower than the maximum offset of data that was already received. Or an endpoint received a RST_STREAM frame containing a different final offset to the one already established.

FRAME_ENCODING_ERROR (0x7): An endpoint received a frame that was badly formatted. For instance, a frame of an unknown type, or an ACK frame that has more acknowledgment ranges than the remainder of the packet could carry.

TRANSPORT_PARAMETER_ERROR (0x8): An endpoint received transport parameters that were badly formatted, included an invalid value, was absent even though it is mandatory, was present though it is forbidden, or is otherwise in error.

VERSION_NEGOTIATION_ERROR (0x9): An endpoint received transport parameters that contained version negotiation parameters that disagreed with the version negotiation that it performed. This error code indicates a potential version downgrade attack.

PROTOCOL_VIOLATION (0xA): An endpoint detected an error with protocol compliance that was not covered by more specific error codes.

INVALID_MIGRATION (0xC): A peer has migrated to a different network when the endpoint had disabled migration.

CRYPTO_ERROR (0x1XX): The cryptographic handshake failed. A range of 256 values is reserved for carrying error codes specific to the cryptographic handshake that is used. Codes for errors occurring when TLS is used for the crypto handshake are described in Section 4.8 of [QUIC-TLS].

See Section 22.3 for details of registering new error codes.

20.1. Application Protocol Error Codes

Application protocol error codes are 16-bit unsigned integers, but the management of application error codes are left to application protocols. Application protocol error codes are used for the RST_STREAM (Section 19.2) and APPLICATION_CLOSE (Section 19.4) frames.

There is no restriction on the use of the 16-bit error code space for application protocols. However, QUIC reserves the error code with a

value of 0 to mean STOPPING. The application error code of STOPPING (0) is used by the transport to cancel a stream in response to receipt of a STOP_SENDING frame.

21. Security Considerations

21.1. Handshake Denial of Service

As an encrypted and authenticated transport QUIC provides a range of protections against denial of service. Once the cryptographic handshake is complete, QUIC endpoints discard most packets that are not authenticated, greatly limiting the ability of an attacker to interfere with existing connections.

Once a connection is established QUIC endpoints might accept some unauthenticated ICMP packets (see Section 14.1.1), but the use of these packets is extremely limited. The only other type of packet that an endpoint might accept is a stateless reset (Section 10.4) which relies on the token being kept secret until it is used.

During the creation of a connection, QUIC only provides protection against attack from off the network path. All QUIC packets contain proof that the recipient saw a preceding packet from its peer.

The first mechanism used is the source and destination connection IDs, which are required to match those set by a peer. Except for an Initial and stateless reset packets, an endpoint only accepts packets that include a destination connection that matches a connection ID the endpoint previously chose. This is the only protection offered for Version Negotiation packets.

The destination connection ID in an Initial packet is selected by a client to be unpredictable, which serves an additional purpose. The packets that carry the cryptographic handshake are protected with a key that is derived from this connection ID and salt specific to the QUIC version. This allows endpoints to use the same process for authenticating packets that they receive as they use after the cryptographic handshake completes. Packets that cannot be authenticated are discarded. Protecting packets in this fashion provides a strong assurance that the sender of the packet saw the Initial packet and understood it.

These protections are not intended to be effective against an attacker that is able to receive QUIC packets prior to the connection being established. Such an attacker can potentially send packets that will be accepted by QUIC endpoints. This version of QUIC attempts to detect this sort of attack, but it expects that endpoints will fail to establish a connection rather than recovering. For the

most part, the cryptographic handshake protocol [QUIC-TLS] is responsible for detecting tampering during the handshake, though additional validation is required for version negotiation (see Section 7.3.3).

Endpoints are permitted to use other methods to detect and attempt to recover from interference with the handshake. Invalid packets may be identified and discarded using other methods, but no specific method is mandated in this document.

21.2. Spoofed ACK Attack

An attacker might be able to receive an address validation token (Section 8) from the server and then release the IP address it used to acquire that token. The attacker may, in the future, spoof this same address (which now presumably addresses a different endpoint), and initiate a 0-RTT connection with a server on the victim's behalf. The attacker can then spoof ACK frames to the server which cause the server to send excessive amounts of data toward the new owner of the IP address.

There are two possible mitigations to this attack. The simplest one is that a server can unilaterally create a gap in packet-number space. In the non-attack scenario, the client will send an ACK frame with the larger value for largest acknowledged. In the attack scenario, the attacker could acknowledge a packet in the gap. If the server sees an acknowledgment for a packet that was never sent, the connection can be aborted.

The second mitigation is that the server can require that acknowledgments for sent packets match the encryption level of the sent packet. This mitigation is useful if the connection has an ephemeral forward-secure key that is generated and used for every new connection. If a packet sent is protected with a forward-secure key, then any acknowledgments that are received for them MUST also be forward-secure protected. Since the attacker will not have the forward-secure key, the attacker will not be able to generate forward-secure protected packets with ACK frames.

21.3. Optimistic ACK Attack

An endpoint that acknowledges packets it has not received might cause a congestion controller to permit sending at rates beyond what the network supports. An endpoint MAY skip packet numbers when sending packets to detect this behavior. An endpoint can then immediately close the connection with a connection error of type PROTOCOL_VIOLATION (see Section 10.3).

21.4. Slowloris Attacks

The attacks commonly known as Slowloris [SLOWLORIS] try to keep many connections to the target endpoint open and hold them open as long as possible. These attacks can be executed against a QUIC endpoint by generating the minimum amount of activity necessary to avoid being closed for inactivity. This might involve sending small amounts of data, gradually opening flow control windows in order to control the sender rate, or manufacturing ACK frames that simulate a high loss rate.

QUIC deployments SHOULD provide mitigations for the Slowloris attacks, such as increasing the maximum number of clients the server will allow, limiting the number of connections a single IP address is allowed to make, imposing restrictions on the minimum transfer speed a connection is allowed to have, and restricting the length of time an endpoint is allowed to stay connected.

21.5. Stream Fragmentation and Reassembly Attacks

An adversarial sender might intentionally send fragments of stream data in order to cause disproportionate receive buffer memory commitment and/or creation of a large and inefficient data structure.

An adversarial receiver might intentionally not acknowledge packets containing stream data in order to force the sender to store the unacknowledged stream data for retransmission.

The attack on receivers is mitigated if flow control windows correspond to available memory. However, some receivers will over-commit memory and advertise flow control offsets in the aggregate that exceed actual available memory. The over-commitment strategy can lead to better performance when endpoints are well behaved, but renders endpoints vulnerable to the stream fragmentation attack.

QUIC deployments SHOULD provide mitigations against stream fragmentation attacks. Mitigations could consist of avoiding over-committing memory, limiting the size of tracking data structures, delaying reassembly of STREAM frames, implementing heuristics based on the age and duration of reassembly holes, or some combination.

21.6. Stream Commitment Attack

An adversarial endpoint can open lots of streams, exhausting state on an endpoint. The adversarial endpoint could repeat the process on a large number of connections, in a manner similar to SYN flooding attacks in TCP.

Normally, clients will open streams sequentially, as explained in Section 2.1. However, when several streams are initiated at short intervals, transmission error may cause STREAM DATA frames opening streams to be received out of sequence. A receiver is obligated to open intervening streams if a higher-numbered stream ID is received. Thus, on a new connection, opening stream 2000001 opens 1 million streams, as required by the specification.

The number of active streams is limited by the concurrent stream limit transport parameter, as explained in Section 2.2. If chosen judiciously, this limit mitigates the effect of the stream commitment attack. However, setting the limit too low could affect performance when applications expect to open large number of streams.

21.7. Explicit Congestion Notification Attacks

An on-path attacker could manipulate the value of ECN codepoints in the IP header to influence the sender's rate. [RFC3168] discusses manipulations and their effects in more detail.

An on-the-side attacker can duplicate and send packets with modified ECN codepoints to affect the sender's rate. If duplicate packets are discarded by a receiver, an off-path attacker will need to race the duplicate packet against the original to be successful in this attack. Therefore, QUIC receivers ignore ECN codepoints set in duplicate packets (see Section 13.3).

21.8. Stateless Reset Oracle

Stateless resets create a possible denial of service attack analogous to a TCP reset injection. This attack is possible if an attacker is able to cause a stateless reset token to be generated for a connection with a selected connection ID. An attacker that can cause this token to be generated can reset an active connection with the same connection ID.

If a packet can be routed to different instances that share a static key, for example by changing an IP address or port, then an attacker can cause the server to send a stateless reset. To defend against this style of denial service, endpoints that share a static key for stateless reset (see Section 10.4.2) MUST be arranged so that packets with a given connection ID always arrive at an instance that has connection state, unless that connection is no longer active.

In the case of a cluster that uses dynamic load balancing, it's possible that a change in load balancer configuration could happen while an active instance retains connection state; even if an instance retains connection state, the change in routing and

resulting stateless reset will result in the connection being terminated. If there is no chance in the packet being routed to the correct instance, it is better to send a stateless reset than wait for connections to time out. However, this is acceptable only if the routing cannot be influenced by an attacker.

22. IANA Considerations

22.1. QUIC Transport Parameter Registry

IANA [SHALL add/has added] a registry for "QUIC Transport Parameters" under a "QUIC Protocol" heading.

The "QUIC Transport Parameters" registry governs a 16-bit space. This space is split into two spaces that are governed by different policies. Values with the first byte in the range 0x00 to 0xfe (in hexadecimal) are assigned via the Specification Required policy [RFC8126]. Values with the first byte 0xff are reserved for Private Use [RFC8126].

Registrations MUST include the following fields:

Value: The numeric value of the assignment (registrations will be between 0x0000 and 0xfeff).

Parameter Name: A short mnemonic for the parameter.

Specification: A reference to a publicly available specification for the value.

The nominated expert(s) verify that a specification exists and is readily accessible. Expert(s) are encouraged to be biased towards approving registrations unless they are abusive, frivolous, or actively harmful (not merely aesthetically displeasing, or architecturally dubious).

The initial contents of this registry are shown in Table 7.

Value	Parameter Name	Specification
0x0000	initial_max_stream_data_bidi_local	Section 18.1
0x0001	initial_max_data	Section 18.1
0x0002	initial_max_bidi_streams	Section 18.1
0x0003	idle_timeout	Section 18.1
0x0004	preferred_address	Section 18.1
0x0005	max_packet_size	Section 18.1
0x0006	stateless_reset_token	Section 18.1
0x0007	ack_delay_exponent	Section 18.1
0x0008	initial_max_uni_streams	Section 18.1
0x0009	disable_migration	Section 18.1
0x000a	initial_max_stream_data_bidi_remote	Section 18.1
0x000b	initial_max_stream_data_uni	Section 18.1
0x000c	max_ack_delay	Section 18.1
0x000d	original_connection_id	Section 18.1

Table 7: Initial QUIC Transport Parameters Entries

22.2. QUIC Frame Type Registry

IANA [SHALL add/has added] a registry for "QUIC Frame Types" under a "QUIC Protocol" heading.

The "QUIC Frame Types" registry governs a 62-bit space. This space is split into three spaces that are governed by different policies. Values between 0x00 and 0x3f (in hexadecimal) are assigned via the Standards Action or IESG Review policies [RFC8126]. Values from 0x40 to 0x3fff operate on the Specification Required policy [RFC8126]. All other values are assigned to Private Use [RFC8126].

Registrations MUST include the following fields:

Value: The numeric value of the assignment (registrations will be between 0x00 and 0x3fff). A range of values MAY be assigned.

Frame Name: A short mnemonic for the frame type.

Specification: A reference to a publicly available specification for the value.

The nominated expert(s) verify that a specification exists and is readily accessible. Specifications for new registrations need to describe the means by which an endpoint might determine that it can send the identified type of frame. An accompanying transport parameter registration (see Section 22.1) is expected for most registrations. The specification needs to describe the format and assigned semantics of any fields in the frame.

Expert(s) are encouraged to be biased towards approving registrations unless they are abusive, frivolous, or actively harmful (not merely aesthetically displeasing, or architecturally dubious).

The initial contents of this registry are tabulated in Table 3.

22.3. QUIC Transport Error Codes Registry

IANA [SHALL add/has added] a registry for "QUIC Transport Error Codes" under a "QUIC Protocol" heading.

The "QUIC Transport Error Codes" registry governs a 16-bit space. This space is split into two spaces that are governed by different policies. Values with the first byte in the range 0x00 to 0xfe (in hexadecimal) are assigned via the Specification Required policy [RFC8126]. Values with the first byte 0xff are reserved for Private Use [RFC8126].

Registrations MUST include the following fields:

Value: The numeric value of the assignment (registrations will be between 0x0000 and 0xfeff).

Code: A short mnemonic for the parameter.

Description: A brief description of the error code semantics, which MAY be a summary if a specification reference is provided.

Specification: A reference to a publicly available specification for the value.

The initial contents of this registry are shown in Table 8. Values from 0xFF00 to 0xFFFF are reserved for Private Use [RFC8126].

Value	Error	Description	Specification
0x0	NO_ERROR	No error	Section 20
0x1	INTERNAL_ERROR	Implementation error	Section 20
0x2	SERVER_BUSY	Server currently busy	Section 20
0x3	FLOW_CONTROL_ERROR	Flow control error	Section 20
0x4	STREAM_ID_ERROR	Invalid stream ID	Section 20
0x5	STREAM_STATE_ERROR	Frame received in invalid stream state	Section 20
0x6	FINAL_OFFSET_ERROR	Change to final stream offset	Section 20
0x7	FRAME_ENCODING_ERROR	Frame encoding error	Section 20
0x8	TRANSPORT_PARAMETER_ERROR	Error in transport parameters	Section 20
0x9	VERSION_NEGOTIATION_ERROR	Version negotiation failure	Section 20
0xA	PROTOCOL_VIOLATION	Generic protocol violation	Section 20
0xC	INVALID_MIGRATION	Violated disabled migration	Section 20

Table 8: Initial QUIC Transport Error Codes Entries

23. References

23.1. Normative References

[PLPMTUD] Mathis, M. and J. Heffner, "Packetization Layer Path MTU Discovery", RFC 4821, DOI 10.17487/RFC4821, March 2007, <https://www.rfc-editor.org/info/rfc4821>.

[PMTUDv4] Mogul, J. and S. Deering, "Path MTU discovery", RFC 1191, DOI 10.17487/RFC1191, November 1990, <https://www.rfc-editor.org/info/rfc1191>.

[PMTUDv6] McCann, J., Deering, S., Mogul, J., and R. Hinden, Ed., "Path MTU Discovery for IP version 6", STD 87, RFC 8201, DOI 10.17487/RFC8201, July 2017, <https://www.rfc-editor.org/info/rfc8201>.

[QUIC-RECOVERY] Iyengar, J., Ed. and I. Swett, Ed., "QUIC Loss Detection and Congestion Control", draft-ietf-quic-recovery-16 (work in progress), October 2018.

[QUIC-TLS] Thomson, M., Ed. and S. Turner, Ed., "Using Transport Layer Security (TLS) to Secure QUIC", draft-ietf-quic-tls-16 (work in progress), October 2018.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3168] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, DOI 10.17487/RFC3168, September 2001, <https://www.rfc-editor.org/info/rfc3168>.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <https://www.rfc-editor.org/info/rfc3629>.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <https://www.rfc-editor.org/info/rfc4086>.

[RFC5119] Edwards, T., "A Uniform Resource Name (URN) Namespace for the Society of Motion Picture and Television Engineers (SMPTE)", RFC 5119, DOI 10.17487/RFC5119, February 2008, <https://www.rfc-editor.org/info/rfc5119>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8311] Black, D., "Relaxing Restrictions on Explicit Congestion Notification (ECN) Experimentation", RFC 8311, DOI 10.17487/RFC8311, January 2018, <https://www.rfc-editor.org/info/rfc8311>.

[TLS13] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

23.2. Informative References

[EARLY-DESIGN] Roskind, J., "QUIC: Multiplexed Transport Over UDP", December 2013, <https://goo.gl/dMVtFi>.

[HTTP2] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <https://www.rfc-editor.org/info/rfc7540>.

[QUIC-INVARIANTS] Thomson, M., "Version-Independent Properties of QUIC", draft-ietf-quic-invariants-03 (work in progress), October 2018.

[RFC2018] Mathis, M., Mahdavi, J., Floyd, S., and A. Romanow, "TCP Selective Acknowledgment Options", RFC 2018, DOI 10.17487/RFC2018, October 1996, <https://www.rfc-editor.org/info/rfc2018>.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <https://www.rfc-editor.org/info/rfc2104>.

[RFC2360] Scott, G., "Guide for Internet Standards Writers", BCP 22, RFC 2360, DOI 10.17487/RFC2360, June 1998, <https://www.rfc-editor.org/info/rfc2360>.

[RFC2406] Kent, S. and R. Atkinson, "IP Encapsulating Security Payload (ESP)", RFC 2406, DOI 10.17487/RFC2406, November 1998, <https://www.rfc-editor.org/info/rfc2406>.

[RFC4787] Audet, F., Ed. and C. Jennings, "Network Address Translation (NAT) Behavioral Requirements for Unicast UDP", BCP 127, RFC 4787, DOI 10.17487/RFC4787, January 2007, <https://www.rfc-editor.org/info/rfc4787>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/info/rfc5869>.

[RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <https://www.rfc-editor.org/info/rfc7301>.

[SLOWLORIS] RSnake Hansen, R., "Welcome to Slowloris...", June 2009, <https://web.archive.org/web/20150315054838/http://ha.ckers.org/slowloris/>.

[SST] Ford, B., "Structured streams", ACM SIGCOMM Computer Communication Review Vol. 37, pp. 361, DOI 10.1145/1282427.1282421, October 2007.

Appendix A. Sample Packet Number Decoding Algorithm

The following pseudo-code shows how an implementation can decode packet numbers after packet number protection has been removed.

```
DecodePacketNumber(largest_pn, truncated_pn, pn_nbits):
   expected_pn  = largest_pn + 1
   pn_win       = 1 << pn_nbits
   pn_hwin      = pn_win / 2
   pn_mask      = pn_win - 1
   // The incoming packet number should be greater than
   // expected_pn - pn_hwin and less than or equal to
   // expected_pn + pn_hwin
   //
   // This means we can't just strip the trailing bits from
   // expected_pn and add the truncated_pn because that might
   // yield a value outside the window.
   //
   // The following code calculates a candidate value and
   // makes sure it's within the packet number window.
   candidate_pn = (expected_pn & ~pn_mask) | truncated_pn
   if candidate_pn <= expected_pn - pn_hwin:
      return candidate_pn + pn_win
   // Note the extra check for underflow when candidate_pn
   // is near zero.
   if candidate_pn > expected_pn + pn_hwin and
      candidate_pn > pn_win:
      return candidate_pn - pn_win
   return candidate_pn
```

Appendix B. Change Log

RFC Editor's Note: Please remove this section prior to publication of a final version of this document.

Issue and pull request numbers are listed with a leading octothorp.

B.1. Since draft-ietf-quic-transport-15

Substantial editorial reorganization; no technical changes.

B.2. Since draft-ietf-quic-transport-14

- Merge ACK and ACK_ECN (#1778, #1801)
- Explicitly communicate max_ack_delay (#981, #1781)
- Validate original connection ID after Retry packets (#1710, #1486, #1793)
- Idle timeout is optional and has no specified maximum (#1765)

- Update connection ID handling; add RETIRE_CONNECTION_ID type (#1464, #1468, #1483, #1484, #1486, #1495, #1729, #1742, #1799, #1821)

- Include a Token in all Initial packets (#1649, #1794)

- Prevent handshake deadlock (#1764, #1824)

B.3. Since draft-ietf-quic-transport-13

- Streams open when higher-numbered streams of the same type open (#1342, #1549)

- Split initial stream flow control limit into 3 transport parameters (#1016, #1542)

- All flow control transport parameters are optional (#1610)

- Removed UNSOLICITED_PATH_RESPONSE error code (#1265, #1539)

- Permit stateless reset in response to any packet (#1348, #1553)

- Recommended defense against stateless reset spoofing (#1386, #1554)

- Prevent infinite stateless reset exchanges (#1443, #1627)

- Forbid processing of the same packet number twice (#1405, #1624)

- Added a packet number decoding example (#1493)

- More precisely define idle timeout (#1429, #1614, #1652)

- Corrected format of Retry packet and prevented looping (#1492, #1451, #1448, #1498)

- Permit 0-RTT after receiving Version Negotiation or Retry (#1507, #1514, #1621)

- Permit Retry in response to 0-RTT (#1547, #1552)

- Looser verification of ECN counters to account for ACK loss (#1555, #1481, #1565)

- Remove frame type field from APPLICATION_CLOSE (#1508, #1528)

B.4. Since draft-ietf-quic-transport-12

- Changes to integration of the TLS handshake (#829, #1018, #1094, #1165, #1190, #1233, #1242, #1252, #1450, #1458)
  - The cryptographic handshake uses CRYPTO frames, not stream 0
  - QUIC packet protection is used in place of TLS record protection
  - Separate QUIC packet number spaces are used for the handshake
  - Changed Retry to be independent of the cryptographic handshake
  - Added NEW_TOKEN frame and Token fields to Initial packet
  - Limit the use of HelloRetryRequest to address TLS needs (like key shares)
- Enable server to transition connections to a preferred address (#560, #1251, #1373)
- Added ECN feedback mechanisms and handling; new ACK_ECN frame (#804, #805, #1372)
- Changed rules and recommendations for use of new connection IDs (#1258, #1264, #1276, #1280, #1419, #1452, #1453, #1465)
- Added a transport parameter to disable intentional connection migration (#1271, #1447)
- Packets from different connection ID can't be coalesced (#1287, #1423)
- Fixed sampling method for packet number encryption; the length field in long headers includes the packet number field in addition to the packet payload (#1387, #1389)
- Stateless Reset is now symmetric and subject to size constraints (#466, #1346)
- Added frame type extension mechanism (#58, #1473)

B.5. Since draft-ietf-quic-transport-11

- Enable server to transition connections to a preferred address (#560, #1251)

- Packet numbers are encrypted (#1174, #1043, #1048, #1034, #850, #990, #734, #1317, #1267, #1079)
- Packet numbers use a variable-length encoding (#989, #1334)
- STREAM frames can now be empty (#1350)

B.6. Since draft-ietf-quic-transport-10

- Swap payload length and packed number fields in long header (#1294)
- Clarified that CONNECTION_CLOSE is allowed in Handshake packet (#1274)
- Spin bit reserved (#1283)
- Coalescing multiple QUIC packets in a UDP datagram (#1262, #1285)
- A more complete connection migration (#1249)
- Refine opportunistic ACK defense text (#305, #1030, #1185)
- A Stateless Reset Token isn't mandatory (#818, #1191)
- Removed implicit stream opening (#896, #1193)
- An empty STREAM frame can be used to open a stream without sending data (#901, #1194)
- Define stream counts in transport parameters rather than a maximum stream ID (#1023, #1065)
- STOP_SENDING is now prohibited before streams are used (#1050)
- Recommend including ACK in Retry packets and allow PADDING (#1067, #882)
- Endpoints now become closing after an idle timeout (#1178, #1179)
- Remove implication that Version Negotiation is sent when a packet of the wrong version is received (#1197)

B.7. Since draft-ietf-quic-transport-09

- Added PATH_CHALLENGE and PATH_RESPONSE frames to replace PING with Data and PONG frame. Changed ACK frame type from 0x0e to 0x0d. (#1091, #725, #1086)

- A server can now only send 3 packets without validating the client address (#38, #1090)
- Delivery order of stream data is no longer strongly specified (#252, #1070)
- Rework of packet handling and version negotiation (#1038)
- Stream 0 is now exempt from flow control until the handshake completes (#1074, #725, #825, #1082)
- Improved retransmission rules for all frame types: information is retransmitted, not packets or frames (#463, #765, #1095, #1053)
- Added an error code for server busy signals (#1137)
- Endpoints now set the connection ID that their peer uses. Connection IDs are variable length. Removed the omit_connection_id transport parameter and the corresponding short header flag. (#1089, #1052, #1146, #821, #745, #821, #1166, #1151)

B.8. Since draft-ietf-quic-transport-08

- Clarified requirements for BLOCKED usage (#65, #924)
- BLOCKED frame now includes reason for blocking (#452, #924, #927, #928)
- GAP limitation in ACK Frame (#613)
- Improved PMTUD description (#614, #1036)
- Clarified stream state machine (#634, #662, #743, #894)
- Reserved versions don't need to be generated deterministically (#831, #931)
- You don't always need the draining period (#871)
- Stateless reset clarified as version-specific (#930, #986)
- initial_max_stream_id_x transport parameters are optional (#970, #971)
- Ack Delay assumes a default value during the handshake (#1007, #1009)
- Removed transport parameters from NewSessionTicket (#1015)

B.9. Since draft-ietf-quic-transport-07

- The long header now has version before packet number (#926, #939)
- Rename and consolidate packet types (#846, #822, #847)
- Packet types are assigned new codepoints and the Connection ID Flag is inverted (#426, #956)
- Removed type for Version Negotiation and use Version 0 (#963, #968)
- Streams are split into unidirectional and bidirectional (#643, #656, #720, #872, #175, #885)
  - Stream limits now have separate uni- and bi-directional transport parameters (#909, #958)
  - Stream limit transport parameters are now optional and default to 0 (#970, #971)
- The stream state machine has been split into read and write (#634, #894)
- Employ variable-length integer encodings throughout (#595)
- Improvements to connection close
  - Added distinct closing and draining states (#899, #871)
  - Draining period can terminate early (#869, #870)
  - Clarifications about stateless reset (#889, #890)
- Address validation for connection migration (#161, #732, #878)
- Clearly defined retransmission rules for BLOCKED (#452, #65, #924)
- negotiated_version is sent in server transport parameters (#710, #959)
- Increased the range over which packet numbers are randomized (#864, #850, #964)

B.10. Since draft-ietf-quic-transport-06

- Replaced FNV-1a with AES-GCM for all "Cleartext" packets (#554)
- Split error code space between application and transport (#485)
- Stateless reset token moved to end (#820)
- 1-RTT-protected long header types removed (#848)
- No acknowledgments during draining period (#852)
- Remove "application close" as a separate close type (#854)
- Remove timestamps from the ACK frame (#841)
- Require transport parameters to only appear once (#792)

B.11. Since draft-ietf-quic-transport-05

- Stateless token is server-only (#726)
- Refactor section on connection termination (#733, #748, #328, #177)
- Limit size of Version Negotiation packet (#585)
- Clarify when and what to ack (#736)
- Renamed STREAM_ID_NEEDED to STREAM_ID_BLOCKED
- Clarify Keep-alive requirements (#729)

B.12. Since draft-ietf-quic-transport-04

- Introduce STOP_SENDING frame, RST_STREAM only resets in one direction (#165)
- Removed GOAWAY; application protocols are responsible for graceful shutdown (#696)
- Reduced the number of error codes (#96, #177, #184, #211)
- Version validation fields can't move or change (#121)
- Removed versions from the transport parameters in a NewSessionTicket message (#547)

- Clarify the meaning of "bytes in flight" (#550)
- Public reset is now stateless reset and not visible to the path (#215)
- Reordered bits and fields in STREAM frame (#620)
- Clarifications to the stream state machine (#572, #571)
- Increased the maximum length of the Largest Acknowledged field in ACK frames to 64 bits (#629)
- truncate_connection_id is renamed to omit_connection_id (#659)
- CONNECTION_CLOSE terminates the connection like TCP RST (#330, #328)
- Update labels used in HKDF-Expand-Label to match TLS 1.3 (#642)

B.13. Since draft-ietf-quic-transport-03

- Change STREAM and RST_STREAM layout
- Add MAX_STREAM_ID settings

B.14. Since draft-ietf-quic-transport-02

- The size of the initial packet payload has a fixed minimum (#267, #472)
- Define when Version Negotiation packets are ignored (#284, #294, #241, #143, #474)
- The 64-bit FNV-1a algorithm is used for integrity protection of unprotected packets (#167, #480, #481, #517)
- Rework initial packet types to change how the connection ID is chosen (#482, #442, #493)
- No timestamps are forbidden in unprotected packets (#542, #429)
- Cryptographic handshake is now on stream 0 (#456)
- Remove congestion control exemption for cryptographic handshake (#248, #476)
- Version 1 of QUIC uses TLS; a new version is needed to use a different handshake protocol (#516)

- STREAM frames have a reduced number of offset lengths (#543, #430)
- Split some frames into separate connection- and stream- level frames (#443)
  - WINDOW_UPDATE split into MAX_DATA and MAX_STREAM_DATA (#450)
  - BLOCKED split to match WINDOW_UPDATE split (#454)
  - Define STREAM_ID_NEEDED frame (#455)
- A NEW_CONNECTION_ID frame supports connection migration without linkability (#232, #491, #496)
- Transport parameters for 0-RTT are retained from a previous connection (#405, #513, #512)
  - A client in 0-RTT no longer required to reset excess streams (#425, #479)
- Expanded security considerations (#440, #444, #445, #448)

B.15. Since draft-ietf-quic-transport-01

- Defined short and long packet headers (#40, #148, #361)
- Defined a versioning scheme and stable fields (#51, #361)
- Define reserved version values for "greasing" negotiation (#112, #278)
- The initial packet number is randomized (#35, #283)
- Narrow the packet number encoding range requirement (#67, #286, #299, #323, #356)
- Defined client address validation (#52, #118, #120, #275)
- Define transport parameters as a TLS extension (#49, #122)
- SCUP and COPT parameters are no longer valid (#116, #117)
- Transport parameters for 0-RTT are either remembered from before, or assume default values (#126)
- The server chooses connection IDs in its final flight (#119, #349, #361)

- The server echoes the Connection ID and packet number fields when sending a Version Negotiation packet (#133, #295, #244)
- Defined a minimum packet size for the initial handshake packet from the client (#69, #136, #139, #164)
- Path MTU Discovery (#64, #106)
- The initial handshake packet from the client needs to fit in a single packet (#338)
- Forbid acknowledgment of packets containing only ACK and PADDING (#291)
- Require that frames are processed when packets are acknowledged (#381, #341)
- Removed the STOP_WAITING frame (#66)
- Don't require retransmission of old timestamps for lost ACK frames (#308)
- Clarified that frames are not retransmitted, but the information in them can be (#157, #298)
- Error handling definitions (#335)
- Split error codes into four sections (#74)
- Forbid the use of Public Reset where CONNECTION_CLOSE is possible (#289)
- Define packet protection rules (#336)
- Require that stream be entirely delivered or reset, including acknowledgment of all STREAM frames or the RST_STREAM, before it closes (#381)
- Remove stream reservation from state machine (#174, #280)
- Only stream 1 does not contribute to connection-level flow control (#204)
- Stream 1 counts towards the maximum concurrent stream limit (#201, #282)
- Remove connection-level flow control exclusion for some streams (except 1) (#246)

- RST_STREAM affects connection-level flow control (#162, #163)
- Flow control accounting uses the maximum data offset on each stream, rather than bytes received (#378)
- Moved length-determining fields to the start of STREAM and ACK (#168, #277)
- Added the ability to pad between frames (#158, #276)
- Remove error code and reason phrase from GOAWAY (#352, #355)
- GOAWAY includes a final stream number for both directions (#347)
- Error codes for RST_STREAM and CONNECTION_CLOSE are now at a consistent offset (#249)
- Defined priority as the responsibility of the application protocol (#104, #303)

B.16. Since draft-ietf-quic-transport-00

- Replaced DIVERSIFICATION_NONCE flag with KEY_PHASE flag
- Defined versioning
- Reworked description of packet and frame layout
- Error code space is divided into regions for each component
- Use big endian for all numeric values

B.17. Since draft-hamilton-quic-transport-protocol-01

- Adopted as base for draft-ietf-quic-tls
- Updated authors/editors list
- Added IANA Considerations section
- Moved Contributors and Acknowledgments to appendices

Acknowledgments

Special thanks are due to the following for helping shape pre-IETF QUIC and its deployment: Chris Bentzel, Misha Efimov, Roberto Peon, Alistair Riddoch, Siddharth Vijayakrishnan, and Assar Westerlund.

This document has benefited immensely from various private discussions and public ones on the quic@ietf.org and proto-quic@chromium.org mailing lists. Our thanks to all.

Contributors

The original authors of this specification were Ryan Hamilton, Jana Iyengar, Ian Swett, and Alyssa Wilk.

The original design and rationale behind this protocol draw significantly from work by Jim Roskind [EARLY-DESIGN]. In alphabetical order, the contributors to the pre-IETF QUIC project at Google 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.

Authors' Addresses

Jana Iyengar (editor)
Fastly

Email: jri.ietf@gmail.com

Martin Thomson (editor)
Mozilla

Email: martin.thomson@gmail.com