

QUIC
Internet-Draft
Intended status: Standards Track
Expires: March 15, 2020

J. Iyengar, Ed.
Fastly
M. Thomson, Ed.
Mozilla
September 12, 2019

QUIC: A UDP-Based Multiplexed and Secure Transport
draft-ietf-quic-transport-23

Abstract

This document defines the core of the QUIC transport protocol. Accompanying documents describe QUIC's loss detection and congestion control and the use of TLS for key negotiation.

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 March 15, 2020.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to [BCP 78](#) and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

| | |
|---|----|
| 1. Introduction | 6 |
| 1.1. Document Structure | 6 |
| 1.2. Terms and Definitions | 8 |
| 1.3. Notational Conventions | 8 |
| 2. Streams | 9 |
| 2.1. Stream Types and Identifiers | 9 |
| 2.2. Sending and Receiving Data | 10 |
| 2.3. Stream Prioritization | 11 |
| 2.4. Required Operations on Streams | 11 |
| 3. Stream States | 12 |
| 3.1. Sending Stream States | 12 |
| 3.2. Receiving Stream States | 14 |
| 3.3. Permitted Frame Types | 17 |
| 3.4. Bidirectional Stream States | 17 |
| 3.5. Solicited State Transitions | 19 |
| 4. Flow Control | 20 |
| 4.1. Data Flow Control | 20 |
| 4.2. Flow Credit Increments | 21 |
| 4.3. Handling Stream Cancellation | 22 |
| 4.4. Stream Final Size | 23 |
| 4.5. Controlling Concurrency | 23 |
| 5. Connections | 24 |
| 5.1. Connection ID | 24 |
| 5.1.1. Issuing Connection IDs | 25 |
| 5.1.2. Consuming and Retiring Connection IDs | 26 |
| 5.2. Matching Packets to Connections | 27 |
| 5.2.1. Client Packet Handling | 28 |
| 5.2.2. Server Packet Handling | 28 |
| 5.3. Life of a QUIC Connection | 29 |
| 5.4. Required Operations on Connections | 29 |
| 6. Version Negotiation | 30 |
| 6.1. Sending Version Negotiation Packets | 30 |
| 6.2. Handling Version Negotiation Packets | 31 |
| 6.2.1. Version Negotiation Between Draft Versions | 31 |
| 6.3. Using Reserved Versions | 31 |
| 7. Cryptographic and Transport Handshake | 32 |

| | | |
|---------|---|----|
| 7.1. | Example Handshake Flows | 33 |
| 7.2. | Negotiating Connection IDs | 34 |
| 7.3. | Transport Parameters | 36 |
| 7.3.1. | Values of Transport Parameters for 0-RTT | 36 |
| 7.3.2. | New Transport Parameters | 38 |
| 7.4. | Cryptographic Message Buffering | 38 |
| 8. | Address Validation | 38 |
| 8.1. | Address Validation During Connection Establishment | 39 |
| 8.1.1. | Address Validation using Retry Packets | 40 |
| 8.1.2. | Address Validation for Future Connections | 40 |
| 8.1.3. | Address Validation Token Integrity | 43 |
| 8.2. | Path Validation | 43 |
| 8.3. | Initiating Path Validation | 44 |
| 8.4. | Path Validation Responses | 44 |
| 8.5. | Successful Path Validation | 44 |
| 8.6. | Failed Path Validation | 45 |
| 9. | Connection Migration | 45 |
| 9.1. | Probing a New Path | 46 |
| 9.2. | Initiating Connection Migration | 47 |
| 9.3. | Responding to Connection Migration | 47 |
| 9.3.1. | Peer Address Spoofing | 48 |
| 9.3.2. | On-Path Address Spoofing | 48 |
| 9.3.3. | Off-Path Packet Forwarding | 49 |
| 9.4. | Loss Detection and Congestion Control | 50 |
| 9.5. | Privacy Implications of Connection Migration | 51 |
| 9.6. | Server's Preferred Address | 52 |
| 9.6.1. | Communicating a Preferred Address | 52 |
| 9.6.2. | Responding to Connection Migration | 52 |
| 9.6.3. | Interaction of Client Migration and Preferred Address | 53 |
| 9.7. | Use of IPv6 Flow-Label and Migration | 53 |
| 10. | Connection Termination | 54 |
| 10.1. | Closing and Draining Connection States | 54 |
| 10.2. | Idle Timeout | 55 |
| 10.3. | Immediate Close | 56 |
| 10.4. | Stateless Reset | 57 |
| 10.4.1. | Detecting a Stateless Reset | 60 |
| 10.4.2. | Calculating a Stateless Reset Token | 61 |
| 10.4.3. | Looping | 62 |
| 11. | Error Handling | 62 |
| 11.1. | Connection Errors | 63 |
| 11.2. | Stream Errors | 64 |
| 12. | Packets and Frames | 64 |
| 12.1. | Protected Packets | 64 |
| 12.2. | Coalescing Packets | 65 |
| 12.3. | Packet Numbers | 66 |
| 12.4. | Frames and Frame Types | 67 |
| 13. | Packetization and Reliability | 70 |
| 13.1. | Packet Processing | 71 |

| | | |
|---------|---|-----|
| 13.2. | Generating Acknowledgements | 71 |
| 13.2.1. | Sending ACK Frames | 71 |
| 13.2.2. | Managing ACK Ranges | 73 |
| 13.2.3. | Receiver Tracking of ACK Frames | 73 |
| 13.2.4. | Limiting ACK Ranges | 73 |
| 13.2.5. | Measuring and Reporting Host Delay | 74 |
| 13.2.6. | ACK Frames and Packet Protection | 74 |
| 13.3. | Retransmission of Information | 75 |
| 13.4. | Explicit Congestion Notification | 77 |
| 13.4.1. | ECN Counts | 77 |
| 13.4.2. | ECN Validation | 78 |
| 14. | Packet Size | 80 |
| 14.1. | Path Maximum Transmission Unit (PMTU) | 81 |
| 14.2. | ICMP Packet Too Big Messages | 82 |
| 14.3. | Datagram Packetization Layer PMTU Discovery | 83 |
| 14.3.1. | PMTU Probes Containing Source Connection ID | 83 |
| 15. | Versions | 83 |
| 16. | Variable-Length Integer Encoding | 84 |
| 17. | Packet Formats | 85 |
| 17.1. | Packet Number Encoding and Decoding | 85 |
| 17.2. | Long Header Packets | 86 |
| 17.2.1. | Version Negotiation Packet | 89 |
| 17.2.2. | Initial Packet | 91 |
| 17.2.3. | 0-RTT | 93 |
| 17.2.4. | Handshake Packet | 95 |
| 17.2.5. | Retry Packet | 96 |
| 17.3. | Short Header Packets | 98 |
| 17.3.1. | Latency Spin Bit | 100 |
| 18. | Transport Parameter Encoding | 101 |
| 18.1. | Reserved Transport Parameters | 102 |
| 18.2. | Transport Parameter Definitions | 102 |
| 19. | Frame Types and Formats | 106 |
| 19.1. | PADDING Frame | 106 |
| 19.2. | PING Frame | 106 |
| 19.3. | ACK Frames | 106 |
| 19.3.1. | ACK Ranges | 108 |
| 19.3.2. | ECN Counts | 110 |
| 19.4. | RESET_STREAM Frame | 111 |
| 19.5. | STOP_SENDING Frame | 111 |
| 19.6. | CRYPTO Frame | 112 |
| 19.7. | NEW_TOKEN Frame | 113 |
| 19.8. | STREAM Frames | 114 |
| 19.9. | MAX_DATA Frame | 115 |
| 19.10. | MAX_STREAM_DATA Frame | 116 |
| 19.11. | MAX_STREAMS Frames | 117 |
| 19.12. | DATA_BLOCKED Frame | 118 |
| 19.13. | STREAM_DATA_BLOCKED Frame | 118 |
| 19.14. | STREAMS_BLOCKED Frames | 119 |

| | | |
|-------------|--|-----|
| 19.15. | NEW_CONNECTION_ID Frame | 119 |
| 19.16. | RETIRE_CONNECTION_ID Frame | 121 |
| 19.17. | PATH_CHALLENGE Frame | 122 |
| 19.18. | PATH_RESPONSE Frame | 123 |
| 19.19. | CONNECTION_CLOSE Frames | 123 |
| 19.20. | Extension Frames | 124 |
| 20. | Transport Error Codes | 124 |
| 20.1. | Application Protocol Error Codes | 126 |
| 21. | Security Considerations | 126 |
| 21.1. | Handshake Denial of Service | 126 |
| 21.2. | Amplification Attack | 127 |
| 21.3. | Optimistic ACK Attack | 127 |
| 21.4. | Slowloris Attacks | 128 |
| 21.5. | Stream Fragmentation and Reassembly Attacks | 128 |
| 21.6. | Stream Commitment Attack | 128 |
| 21.7. | Peer Denial of Service | 129 |
| 21.8. | Explicit Congestion Notification Attacks | 129 |
| 21.9. | Stateless Reset Oracle | 130 |
| 21.10. | Version Downgrade | 130 |
| 21.11. | Targeted Attacks by Routing | 130 |
| 22. | IANA Considerations | 131 |
| 22.1. | QUIC Transport Parameter Registry | 131 |
| 22.2. | QUIC Frame Type Registry | 132 |
| 22.3. | QUIC Transport Error Codes Registry | 133 |
| 23. | References | 135 |
| 23.1. | Normative References | 136 |
| 23.2. | Informative References | 137 |
| Appendix A. | Sample Packet Number Decoding Algorithm | 139 |
| Appendix B. | Change Log | 140 |
| B.1. | Since draft-ietf-quic-transport-22 | 140 |
| B.2. | Since draft-ietf-quic-transport-21 | 141 |
| B.3. | Since draft-ietf-quic-transport-20 | 141 |
| B.4. | Since draft-ietf-quic-transport-19 | 142 |
| B.5. | Since draft-ietf-quic-transport-18 | 142 |
| B.6. | Since draft-ietf-quic-transport-17 | 143 |
| B.7. | Since draft-ietf-quic-transport-16 | 144 |
| B.8. | Since draft-ietf-quic-transport-15 | 145 |
| B.9. | Since draft-ietf-quic-transport-14 | 145 |
| B.10. | Since draft-ietf-quic-transport-13 | 145 |
| B.11. | Since draft-ietf-quic-transport-12 | 146 |
| B.12. | Since draft-ietf-quic-transport-11 | 147 |
| B.13. | Since draft-ietf-quic-transport-10 | 147 |
| B.14. | Since draft-ietf-quic-transport-09 | 148 |
| B.15. | Since draft-ietf-quic-transport-08 | 148 |
| B.16. | Since draft-ietf-quic-transport-07 | 149 |
| B.17. | Since draft-ietf-quic-transport-06 | 150 |
| B.18. | Since draft-ietf-quic-transport-05 | 150 |
| B.19. | Since draft-ietf-quic-transport-04 | 151 |

| | |
|---|-----|
| B.20. Since draft-ietf-quic-transport-03 | 151 |
| B.21. Since draft-ietf-quic-transport-02 | 152 |
| B.22. Since draft-ietf-quic-transport-01 | 153 |
| B.23. Since draft-ietf-quic-transport-00 | 155 |
| B.24. Since draft-hamilton-quic-transport-protocol-01 | 155 |
| Acknowledgments | 155 |
| Contributors | 155 |
| Authors' Addresses | 156 |

1. Introduction

QUIC is a multiplexed and secure general-purpose transport protocol that provides:

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

QUIC uses UDP as a substrate to avoid requiring changes to legacy client operating systems and middleboxes. QUIC authenticates all of its headers and encrypts most of the data it exchanges, including its signaling, to avoid incurring a dependency on middleboxes.

1.1. Document Structure

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

- o 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.
- o 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 a new network path,
 - * [Section 10](#) lists the options for terminating an open connection, and
 - * [Section 11](#) provides general guidance for error handling.
- o 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 data, and
 - * [Section 14](#) specifies rules for managing the size of packets.
 - o Finally, encoding details of QUIC protocol elements are described in:
 - * [Section 15](#) (Versions),
 - * [Section 16](#) (Integer Encoding),
 - * [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 for key negotiation [[QUIC-TLS](#)].

This document defines QUIC version 1, which conforms to the protocol invariants in [[QUIC-INVARIANTS](#)].

1.2. Terms and Definitions

The keywords "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.

Commonly used terms in the document are described below.

QUIC: The transport protocol described by this document. QUIC is a name, not an acronym.

QUIC packet: A complete processable unit of QUIC that can be encapsulated in a UDP datagram. Multiple QUIC packets can be encapsulated in a single UDP datagram.

Endpoint: An entity that can participate in a QUIC connection by generating, receiving, and processing QUIC packets. There are only two types of endpoint in QUIC: client and server.

Client: The endpoint initiating a QUIC connection.

Server: The endpoint accepting incoming QUIC connections.

Connection ID: An opaque identifier that is used to identify a QUIC connection at an endpoint. Each endpoint sets a value for its peer to include in packets sent towards the endpoint.

Stream: A unidirectional or bidirectional channel of ordered bytes within a QUIC connection. A QUIC connection can carry multiple simultaneous streams.

Application: An entity that uses QUIC to send and receive data.

1.3. Notational Conventions

Packet and frame diagrams in this document 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 to an application. Streams can be unidirectional or bidirectional. An alternative view of QUIC unidirectional streams is a "message" abstraction of practically unlimited length.

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 19.8](#)) can open, carry data for, and close a stream. Streams can also be long-lived and can last the entire duration of a connection.

Streams can be created by either endpoint, can concurrently send data interleaved with other streams, and can be cancelled. QUIC does not provide any means of ensuring ordering between bytes on different streams.

QUIC allows for an arbitrary number of streams to operate concurrently and for an arbitrary amount of data to be sent on any stream, subject to flow control constraints (see [Section 4](#)) and stream limits.

2.1. Stream Types and Identifiers

Streams can be unidirectional or bidirectional. 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.

Streams are identified within a connection by a numeric value, referred to as the stream ID. A stream ID is a 62-bit integer (0 to $2^{62}-1$) that is unique for all streams on a connection. Stream IDs are encoded as variable-length integers (see [Section 16](#)). A QUIC endpoint MUST NOT reuse a stream ID within a connection.

The least significant bit (0x1) of the stream ID identifies the initiator of the stream. Client-initiated streams have even-numbered stream IDs (with the bit set to 0), and server-initiated streams have odd-numbered stream IDs (with the bit set to 1).

The second least significant bit (0x2) of the stream ID distinguishes between bidirectional streams (with the bit set to 0) and unidirectional streams (with the bit set to 1).

The least significant two bits from a stream ID therefore identify a stream as one of four types, as summarized in Table 1.

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

Table 1: Stream ID Types

Within each type, streams are created with numerically increasing stream IDs. A stream ID that is used out of order results in all streams of that type with lower-numbered stream IDs also being opened.

The first bidirectional stream opened by the client has a stream ID of 0.

2.2. Sending and Receiving Data

STREAM frames ([Section 19.8](#)) encapsulate data sent by an application. An endpoint uses the Stream ID and Offset fields in STREAM frames to place data in order.

Endpoints **MUST** be able to deliver stream data to an application as an ordered byte-stream. Delivering an ordered byte-stream requires that an endpoint buffer any data that is received out of order, up to the advertised flow control limit.

QUIC makes no specific allowances for delivery of stream data out of order. However, implementations **MAY** choose to offer the ability to deliver data out of order to a receiving application.

An endpoint could receive data for a stream at the same stream offset multiple times. Data that has already been received can be discarded. The data at a given offset **MUST NOT** change if it is sent multiple times; an endpoint **MAY** treat receipt of different data at

the same offset within a stream as a connection error of type `PROTOCOL_VIOLATION`.

Streams are an ordered byte-stream abstraction with no other structure visible to QUIC. `STREAM` frame boundaries are not expected to be preserved when data is transmitted, retransmitted after packet loss, or delivered to the application at a receiver.

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

2.3. Stream Prioritization

Stream multiplexing can have a significant effect on application performance if resources allocated to streams are correctly prioritized.

QUIC does not provide a mechanism for exchanging prioritization information. Instead, it relies on receiving priority information from the application that uses QUIC.

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

2.4. Required Operations on Streams

There are certain operations which an application **MUST** be able to perform when interacting with QUIC streams. This document does not specify an API, but any implementation of this version of QUIC **MUST** expose the ability to perform the operations described in this section on a QUIC stream.

On the sending part of a stream, application protocols need to be able to:

- o write data, understanding when stream flow control credit ([Section 4.1](#)) has successfully been reserved to send the written data
- o end the stream (clean termination), resulting in a `STREAM` frame ([Section 19.8](#)) with the `FIN` bit set; and
- o reset the stream (abrupt termination), resulting in a `RESET_STREAM` frame ([Section 19.4](#)), even if the stream was already ended.

On the receiving part of a stream, application protocols need to be able to:

- o read data
- o abort reading of the stream and request closure, possibly resulting in a STOP_SENDING frame ([Section 19.5](#))

Applications also need to be informed of state changes on streams, including when the peer has opened or reset a stream, when a peer aborts reading on a stream, when new data is available, and when data can or cannot be written to the stream due to flow control.

3. Stream States

This section describes streams in terms of their send or receive components. Two state machines are described: one for the streams on which an endpoint transmits data ([Section 3.1](#)), and another for streams on 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 MUST open streams of the same type in increasing order of stream ID.

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. Sending Stream States

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

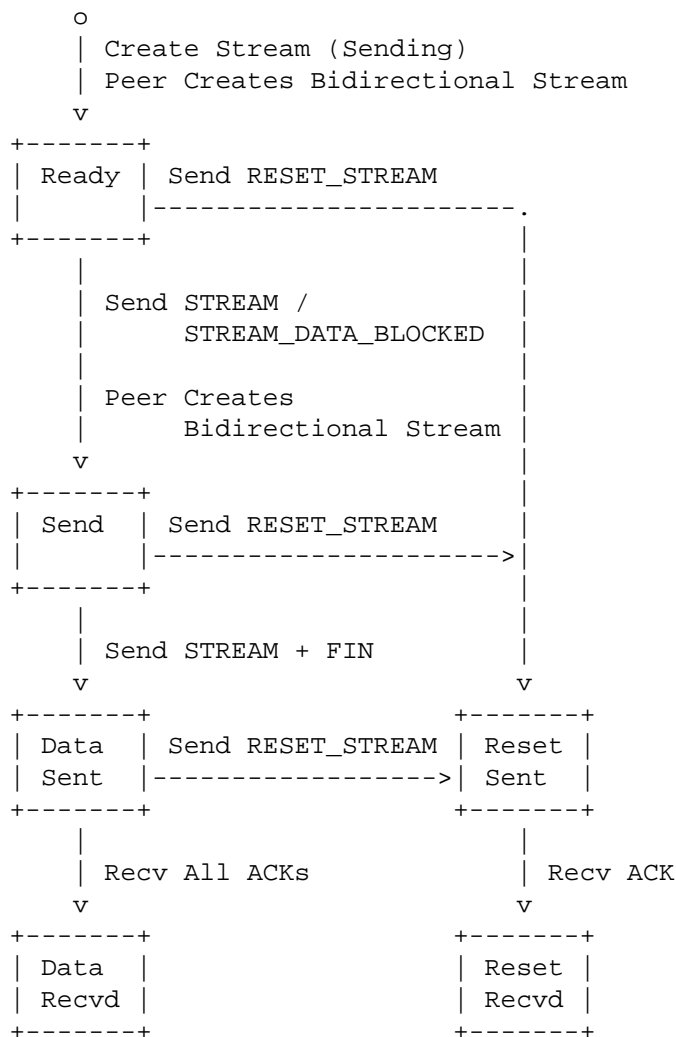


Figure 1: States for Sending Parts of Streams

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. 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_DATA_BLOCKED frame causes a sending part of a stream to enter the "Send" state. An implementation might choose to defer allocating a stream ID to a stream until it sends the first STREAM 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 ([Section 3.2](#)).

In the "Send" state, an endpoint transmits - and retransmits as necessary - stream data in STREAM frames. The endpoint respects the flow control limits set by its peer, and continues to accept and process MAX_STREAM_DATA frames. An endpoint in the "Send" state generates STREAM_DATA_BLOCKED frames if it is blocked from sending by stream or connection flow control limits [Section 4.1](#).

After the application indicates that all stream data has been sent and a STREAM frame containing the FIN bit is sent, the sending part of the stream enters the "Data Sent" state. From this state, the endpoint only retransmits stream data as necessary. The endpoint does not need to check flow control limits or send STREAM_DATA_BLOCKED frames for a stream in this state. MAX_STREAM_DATA frames might be received until the peer receives the final stream offset. The endpoint can safely ignore any MAX_STREAM_DATA frames it receives from its peer for a stream in this state.

Once all stream data has been successfully acknowledged, the sending part of the 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. Alternatively, an endpoint might receive a STOP_SENDING frame from its peer. In either case, the endpoint sends a RESET_STREAM frame, which causes the stream to enter the "Reset Sent" state.

An endpoint MAY send a RESET_STREAM as the first frame that mentions a stream; this causes the sending part of that stream to open and then immediately transition to the "Reset Sent" state.

Once a packet containing a RESET_STREAM has been acknowledged, the sending part of the stream enters the "Reset Recvd" state, which is a terminal state.

[3.2](#). Receiving Stream States

Figure 2 shows the states for the part of a stream that receives data from a peer. The states for a receiving part of a stream mirror only some of the states of the sending part of the stream at the peer. The receiving part of a stream does not track states on the sending

part that cannot be observed, such as the "Ready" state. Instead, the receiving part of a stream tracks the delivery of data to the application, some of which cannot be observed by the sender.

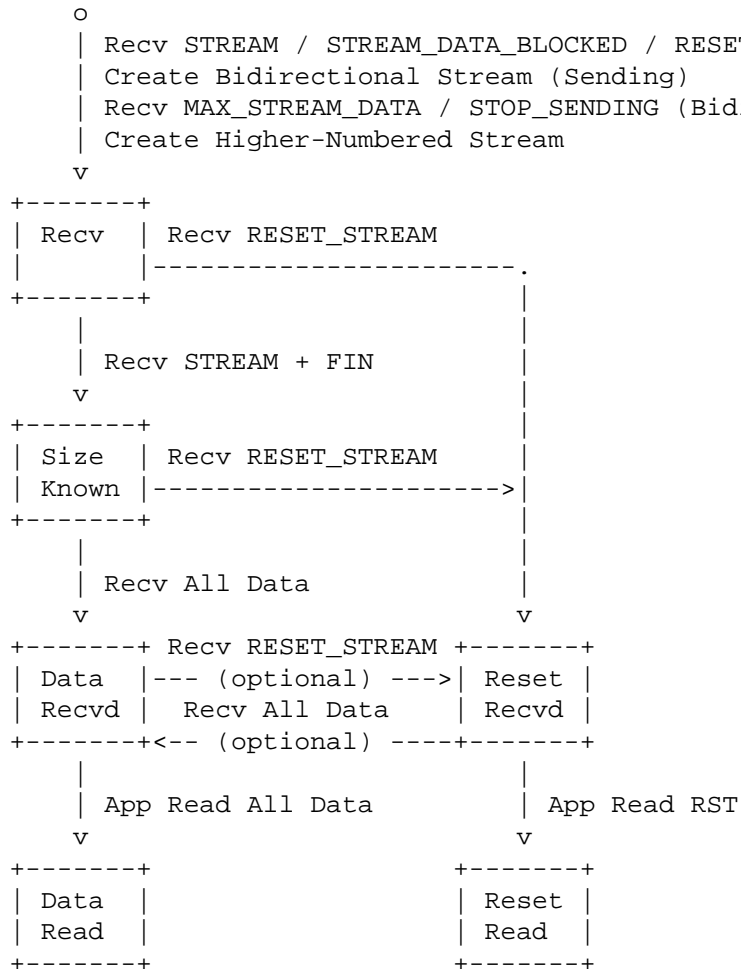


Figure 2: States for Receiving Parts of Streams

The receiving part of a stream initiated by a peer (types 1 and 3 for a client, or 0 and 2 for a server) is created when the first STREAM, STREAM_DATA_BLOCKED, or RESET_STREAM is received for that stream. For bidirectional streams initiated by a peer, receipt of a MAX_STREAM_DATA or STOP_SENDING frame for the sending part of the stream also creates the receiving part. The initial state for the receiving part of stream is "Recv".

The receiving part of a 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.

An endpoint opens a bidirectional stream when a MAX_STREAM_DATA or STOP_SENDING frame is received from the peer for that stream. Receiving a MAX_STREAM_DATA frame for an unopened stream indicates that the remote peer has opened the stream and is providing flow control credit. Receiving a STOP_SENDING frame for an unopened stream indicates that the remote peer no longer wishes to receive data on this stream. Either frame might arrive before a STREAM or STREAM_DATA_BLOCKED frame if packets are lost or reordered.

Before a stream is created, all streams of the same type with lower-numbered stream IDs MUST be created. This ensures that the creation order for streams is consistent on both endpoints.

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

When a STREAM frame with a FIN bit is received, the final size of the stream is known (see [Section 4.4](#)). The receiving part of the stream then 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 receiving part enters the "Data Recvd" state. This might happen as a result of receiving the same STREAM frame that causes the transition to "Size Known". After all data has been received, any STREAM or STREAM_DATA_BLOCKED frames for the stream can be discarded.

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

Receiving a RESET_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 RESET_STREAM is received (that is, from the "Data Recvd" state). Similarly, it is possible for remaining stream data to arrive after receiving a

RESET_STREAM frame (the "Reset Recvd" state). An implementation is free to manage this situation as it chooses.

Sending RESET_STREAM means that an endpoint cannot guarantee delivery of stream data; however there is no requirement that stream data not be delivered if a RESET_STREAM is received. An implementation MAY interrupt delivery of stream data, discard any data that was not consumed, and signal the receipt of the RESET_STREAM. A RESET_STREAM signal might be suppressed or withheld if stream data is completely received and is buffered to be read by the application. If the RESET_STREAM is suppressed, the receiving part of the stream remains in "Data Recvd".

Once the application receives the signal indicating that the stream was reset, the receiving part of the 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 19.8), STREAM_DATA_BLOCKED (Section 19.13), and RESET_STREAM (Section 19.4).

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_DATA_BLOCKED after sending a RESET_STREAM; that is, in the terminal states and in the "Reset Sent" state. A receiver could receive any of these three frames in any state, due to the possibility of delayed delivery of packets carrying them.

The receiver of a stream sends MAX_STREAM_DATA (Section 19.10) and STOP_SENDING frames (Section 19.5).

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 RESET_STREAM frame; that is states other than "Reset Recvd" or "Reset Read". However there is little value in sending a STOP_SENDING frame in the "Data Recvd" state, since all stream data has been received. A sender could receive either of these two frames in any state as a result of delayed delivery of packets.

3.4. Bidirectional Stream States

A bidirectional stream is composed of sending and receiving parts. Implementations may represent states of the bidirectional stream as composites of sending and receiving stream states. The simplest model presents the stream as "open" when either sending or receiving

parts are in a non-terminal state and "closed" when both sending and receiving streams are in terminal states.

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 sending or receiving parts of 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.

| Sending Part | Receiving Part | 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) |
| 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 receiving part of the stream is in the "Recv" state without yet having received any frames.

3.5. Solicited State Transitions

If an application is no longer interested in the data it is receiving on a stream, it can abort reading the stream and specify an application error code.

If the stream is in the "Recv" or "Size Known" states, the transport SHOULD signal this by sending a STOP_SENDING frame 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 it is not a guarantee that incoming data will be ignored.

STREAM frames received after sending STOP_SENDING are still counted toward connection and stream flow control, even though these frames can be discarded upon receipt.

A STOP_SENDING frame requests that the receiving endpoint send a RESET_STREAM frame. An endpoint that receives a STOP_SENDING frame MUST send a RESET_STREAM frame if the stream is in the Ready or Send state. If the stream is in the Data Sent state and any outstanding data is declared lost, an endpoint SHOULD send a RESET_STREAM frame in lieu of a retransmission.

An endpoint SHOULD copy the error code from the STOP_SENDING frame to the RESET_STREAM frame it sends, but MAY use any application error code. The endpoint that sends a STOP_SENDING frame MAY ignore the error code carried in any RESET_STREAM frame it receives.

If the STOP_SENDING frame is received on a stream that is already in the "Data Sent" state, an endpoint that wishes to cease retransmission of previously-sent STREAM frames on that stream MUST first send a RESET_STREAM frame.

STOP_SENDING SHOULD only be sent for a stream that has not been reset by the peer. 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 RESET_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.

An endpoint that wishes to terminate both directions of a bidirectional stream can terminate one direction by sending a RESET_STREAM, and it can encourage prompt termination in the opposite direction by sending a STOP_SENDING frame.

4. Flow Control

It is necessary to limit the amount of data that a receiver could buffer, to prevent a fast sender from overwhelming a slow receiver, or to prevent a malicious sender from consuming a large amount of memory at a receiver. To enable a receiver to limit memory commitment to a connection and to apply back pressure on the sender, streams are flow controlled both individually and as an aggregate. A QUIC receiver controls the maximum amount of data the sender can send on a stream at any time, as described in [Section 4.1](#) and [Section 4.2](#).

Similarly, to limit concurrency within a connection, a QUIC endpoint controls the maximum cumulative number of streams that its peer can initiate, as described in [Section 4.5](#).

Data sent in CRYPTO frames is not flow controlled in the same way as stream data. 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.

4.1. Data Flow Control

QUIC employs a credit-based flow-control scheme similar to that in HTTP/2 [\[HTTP2\]](#), where a receiver advertises the number of bytes it is prepared to receive on a given stream and for the entire connection. This leads to two levels of data flow control in QUIC:

- o Stream flow control, which prevents a single stream from consuming the entire receive buffer for a connection by limiting the amount of data that can be sent on any stream.
- o Connection flow control, which prevents senders from exceeding a receiver's buffer capacity for the connection, by limiting the total bytes of stream data sent in STREAM frames on all streams.

A receiver sets initial credits for all streams by sending transport parameters during the handshake ([Section 7.3](#)). A receiver sends MAX_STREAM_DATA ([Section 19.10](#)) or MAX_DATA ([Section 19.9](#)) frames to the sender to advertise additional credit.

A receiver advertises credit for a stream by sending a MAX_STREAM_DATA frame with the Stream ID field set appropriately. A MAX_STREAM_DATA frame indicates the maximum absolute byte offset of a stream. 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.

A receiver advertises credit for a connection by sending a `MAX_DATA` frame, which indicates the maximum of the sum of the absolute byte offsets of all streams. A receiver maintains a cumulative sum of bytes received on all streams, which is 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 can advertise a larger offset by sending `MAX_STREAM_DATA` or `MAX_DATA` frames. Once a receiver advertises an offset, it MAY advertise a smaller offset, but this has no effect.

A receiver MUST close the connection with a `FLOW_CONTROL_ERROR` error ([Section 11](#)) if the sender violates the advertised connection or stream data limits.

A sender MUST ignore any `MAX_STREAM_DATA` or `MAX_DATA` frames that do not increase flow control limits.

If a sender runs out of flow control credit, it will be unable to send new data and is considered blocked. A sender SHOULD send a `STREAM_DATA_BLOCKED` or `DATA_BLOCKED` frame 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 sender SHOULD NOT send multiple `STREAM_DATA_BLOCKED` or `DATA_BLOCKED` frames for the same data limit, unless the original frame is determined to be lost. Another `STREAM_DATA_BLOCKED` or `DATA_BLOCKED` frame can be sent after the data limit is increased.

4.2. Flow Credit Increments

This document leaves when and how many bytes to advertise in a `MAX_STREAM_DATA` or `MAX_DATA` frame 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 can use an autotuning mechanism to tune the frequency and amount of advertised additional credit based on a round-trip time estimate and the rate at which the receiving application consumes

data, similar to common TCP implementations. As an optimization, sending frames related to flow control only when there are other frames to send or when a peer is blocked ensures that flow control doesn't cause extra packets to be sent.

If a sender runs out of flow control credit, it will be unable to send new data and is considered blocked. It is generally considered best to not let the sender become blocked. 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 receiver MUST NOT wait for a STREAM_DATA_BLOCKED or DATA_BLOCKED frame before sending MAX_STREAM_DATA or MAX_DATA, since doing so will mean that a sender will be blocked for at least an entire round trip, and potentially for longer if the peer chooses to not send STREAM_DATA_BLOCKED or DATA_BLOCKED frames.

4.3. Handling Stream Cancellation

Endpoints need to eventually agree on the amount of flow control credit that has been consumed, to avoid either exceeding flow control limits or deadlocking.

On receipt of a RESET_STREAM frame, an endpoint will tear down state for the matching stream and ignore further data arriving on that stream. Without the offset included in RESET_STREAM, the two endpoints could disagree on the number of bytes that count towards connection flow control.

To remedy this issue, a RESET_STREAM frame ([Section 19.4](#)) includes the final size of data sent on the stream. On receiving a RESET_STREAM frame, a receiver definitively knows how many bytes were sent on that stream before the RESET_STREAM frame, and the receiver MUST use the final size of the stream to account for all bytes sent on the stream in its connection level flow controller.

RESET_STREAM terminates one direction of a stream abruptly. For a bidirectional stream, RESET_STREAM has no effect on data flow in the opposite direction. Both endpoints MUST maintain flow control state for the stream in the unterminated direction until that direction enters a terminal state, or until one of the endpoints sends CONNECTION_CLOSE.

4.4. Stream Final Size

The final size is the amount of flow control credit that is consumed by a stream. Assuming that every contiguous byte on the stream was sent once, the final size is the number of bytes sent. More generally, this is one higher than the offset of the byte with the largest offset sent on the stream, or zero if no bytes were sent.

For a stream that is reset, the final size is carried explicitly in a RESET_STREAM frame. Otherwise, the final size is the offset plus the length of a STREAM frame marked with a FIN flag, or 0 in the case of incoming unidirectional streams.

An endpoint will know the final size for a stream when the receiving part of the stream enters the "Size Known" or "Reset Recvd" state ([Section 3](#)).

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

Once a final size for a stream is known, it cannot change. If a RESET_STREAM or STREAM frame is received indicating a change in the final size for the stream, an endpoint SHOULD respond with a FINAL_SIZE_ERROR error (see [Section 11](#)). A receiver SHOULD treat receipt of data at or beyond the final size as a FINAL_SIZE_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 size state for closed streams, which could mean a significant state commitment.

4.5. Controlling Concurrency

An endpoint limits the cumulative number of incoming streams a peer can open. Only streams with a stream ID less than $(\text{max_stream} * 4 + \text{initial_stream_id_for_type})$ can be opened (see [Table 5](#)). Initial limits are set in the transport parameters (see [Section 18.2](#)) and subsequently limits are advertised using MAX_STREAMS frames ([Section 19.11](#)). Separate limits apply to unidirectional and bidirectional streams.

If a max_streams transport parameter or MAX_STREAMS frame is received with a value greater than 2^{60} , this would allow a maximum stream ID that cannot be expressed as a variable-length integer (see [Section 16](#)). If either is received, the connection MUST be closed immediately with a connection error of type STREAM_LIMIT_ERROR (see [Section 10.3](#)).

Endpoints MUST NOT exceed the limit set by their peer. An endpoint that receives a frame with a stream ID exceeding the limit it has sent MUST treat this as a connection error of type `STREAM_LIMIT_ERROR` ([Section 11](#)).

Once a receiver advertises a stream limit using the `MAX_STREAMS` frame, advertising a smaller limit has no effect. A receiver MUST ignore any `MAX_STREAMS` frame that does not increase the stream limit.

As with stream and connection flow control, this document leaves when and how many streams to advertise to a peer via `MAX_STREAMS` to implementations. Implementations might choose to increase limits as streams close to keep the number of streams available to peers roughly consistent.

An endpoint that is unable to open a new stream due to the peer's limits SHOULD send a `STREAMS_BLOCKED` frame ([Section 19.14](#)). This signal is considered useful for debugging. An endpoint MUST NOT wait to receive this signal before advertising additional credit, since doing so will mean that the peer will be blocked for at least an entire round trip, and potentially for longer if the peer chooses to not send `STREAMS_BLOCKED` frames.

5. Connections

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 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) 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 by an external observer (that is, one that does not cooperate with the issuer) 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 17.2.1](#)) 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 19.15](#)).

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 19.15](#)). 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 Retry 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 19.16](#)).

An endpoint SHOULD ensure that its peer has a sufficient number of available and unused connection IDs. Endpoints store received connection IDs for future use and advertise the number of connection IDs they are willing to store with the `active_connection_id_limit` transport parameter. An endpoint SHOULD NOT provide more connection IDs than the peer's limit.

An endpoint SHOULD supply a new connection ID when it receives a packet with a previously unused connection ID or when the peer retires one, unless providing the new connection ID would exceed the peer's limit. An endpoint MAY limit the frequency or the total number of connection IDs issued for each connection to avoid the risk of running out of connection IDs; see [Section 10.4.2](#).

An endpoint that initiates migration and requires non-zero-length connection IDs SHOULD ensure that the pool of connection IDs available to its peer allows the peer to use a new connection ID on migration, as the peer will close the connection if the pool is exhausted.

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. Sending a `RETIRE_CONNECTION_ID` frame indicates that the connection ID will not be used again and requests that the peer replace it with a new connection ID using a `NEW_CONNECTION_ID` frame.

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.

An endpoint can request that its peer retire connection IDs by sending a NEW_CONNECTION_ID frame with an increased Retire Prior To field. Upon receipt, the peer SHOULD retire the corresponding connection IDs and send the corresponding RETIRE_CONNECTION_ID frames in a timely manner. Failing to do so can cause packets to be delayed, lost, or cause the original endpoint to send a stateless reset in response to a connection ID it can no longer route correctly.

An endpoint MAY discard a connection ID for which retirement has been requested once an interval of no less than 3 PTO has elapsed since an acknowledgement is received for the NEW_CONNECTION_ID frame requesting that retirement. Subsequent incoming packets using that connection ID could elicit a response with the corresponding stateless reset token.

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 SHOULD either reject connection attempts that use the same addresses as existing connections, or use a non-zero-length Destination Connection ID so that packets can be correctly attributed to connections.

Endpoints can send a Stateless Reset ([Section 10.4](#)) for any packets that cannot be attributed to an existing connection. A stateless reset allows a peer to more quickly identify when a connection becomes unusable.

Packets that are matched to an existing connection are discarded if the packets are inconsistent with the state of that connection. For

example, packets are discarded if they indicate a different protocol version than that of the connection, or if the removal of packet protection is unsuccessful once the expected keys are available.

Invalid packets without packet protection, such as Initial, Retry, or Version Negotiation, MAY be discarded. An endpoint MUST generate a connection error if it commits changes to state before discovering an error.

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, a client might receive packets for a connection that are encrypted with a key it has not yet computed. The client 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. Otherwise, servers MUST drop packets that specify unsupported versions.

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.

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 not able to send 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.

5.4. Required Operations on Connections

There are certain operations which an application MUST be able to perform when interacting with the QUIC transport. This document does not specify an API, but any implementation of this version of QUIC MUST expose the ability to perform the operations described in this section on a QUIC connection.

When implementing the client role, applications need to be able to:

- o open a connection, which begins the exchange described in [Section 7](#);
- o enable 0-RTT; and
- o be informed when 0-RTT has been accepted or rejected by a server.

When implementing the server role, applications need to be able to:

- o listen for incoming connections, which prepares for the exchange described in [Section 7](#);
- o if Early Data is supported, embed application-controlled data in the TLS resumption ticket sent to the client; and
- o if Early Data is supported, retrieve application-controlled data from the client's resumption ticket and enable rejecting Early Data based on that information.

In either role, applications need to be able to:

- o configure minimum values for the initial number of permitted streams of each type, as communicated in the transport parameters ([Section 7.3](#));
- o control resource allocation of various types, including flow control and the number of permitted streams of each type;
- o identify whether the handshake has completed successfully or is still ongoing
- o keep a connection from silently closing, either by generating PING frames ([Section 19.2](#)) or by requesting that the transport send additional frames before the idle timeout expires ([Section 10.2](#)); and
- o immediately close ([Section 10.3](#)) the connection.

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 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 17.2.1](#)). This includes a list of versions that the server will accept. An endpoint MUST NOT send a Version Negotiation packet in response to receiving a Version Negotiation packet.

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. As a result, the client discards all state for the connection and does not send any more packets on the connection.

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 a client receives a Version Negotiation packet, it MUST abandon the current connection attempt. Version Negotiation packets are designed to allow future versions of QUIC to negotiate the version in use between endpoints. Future versions of QUIC might change how implementations that support multiple versions of QUIC react to Version Negotiation packets when attempting to establish a connection using this version. How to perform version negotiation is left as future work defined by future versions of QUIC. In particular, that future work will need to ensure robustness against version downgrade attacks [Section 21.10](#).

6.2.1. Version Negotiation Between Draft Versions

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

When a draft implementation receives a Version Negotiation packet, it MAY use it to attempt a new connection with one of the versions listed in the packet, instead of abandoning the current connection attempt [Section 6.2](#).

The client MUST check 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 new connection using that version. The new connection MUST use a new random Destination Connection ID different from the one it had previously sent.

Note that this mechanism does not protect against downgrade attacks and MUST NOT be used outside of draft implementations.

6.3. Using Reserved Versions

For a server to use a new version in the future, clients need to correctly handle unsupported versions. To help ensure this, a server SHOULD include a version that is reserved for forcing version

negotiation (0x?a?a?a as defined in [Section 15](#)) when 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.

A client MAY send a packet using a version that is reserved for forcing version negotiation. 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 19.6](#) to transmit the cryptographic handshake. Version 0x00000001 of QUIC uses TLS 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 is used to encrypt as much of the handshake protocol as possible. The cryptographic handshake MUST provide the following properties:

- o 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
- o authenticated values for transport parameters of both endpoints, and confidentiality protection for server transport parameters (see [Section 7.3](#))
- o 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 byte QUIC packet payload. This includes overheads that reduce the space available to the cryptographic handshake protocol.

An endpoint can verify support for Explicit Congestion Notification (ECN) in the first packets it sends, as described in [Section 13.4.2](#).

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.

Endpoints MUST explicitly negotiate an application protocol. This avoids situations where there is a disagreement about the protocol that is in use.

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 address validation exchanges are complete, the cryptographic handshake is used to agree on cryptographic keys. The cryptographic handshake is carried in Initial ([Section 17.2.2](#)) and Handshake ([Section 17.2.4](#)) packets.

Figure 3 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.

Client

Server

Initial[0]: CRYPTO[CH] ->

```

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

```

```

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

```

```

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

```

Figure 3: Example 1-RTT Handshake

Figure 4 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.

Client

Server

```

Initial[0]: CRYPTO[CH]
0-RTT[0]: STREAM[0, "..."] ->

```

```

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

```

```

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

```

```

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

```

Figure 4: 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 that has not previously received an Initial or Retry packet from the server, it populates the Destination Connection ID field with an unpredictable value. This MUST be at least 8 bytes 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 SCID Len field to indicate the length.

The first flight of 0-RTT packets use the same Destination and Source Connection ID values as the client's first Initial.

Upon 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, including any subsequent 0-RTT packets. That means that a client might change the Destination Connection ID twice during connection establishment, once in response to a Retry and once in response to the first Initial packet from the server. 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.

Definitions for each of the defined transport parameters are included in [Section 18.2](#).

An endpoint **MUST** treat receipt of a transport parameter with an invalid value as a connection error of type `TRANSPORT_PARAMETER_ERROR`.

An endpoint **MUST NOT** send a parameter more than once in a given transport parameters extension. An endpoint **SHOULD** 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 18.2](#)) if it sent a Retry packet to enable validation of the Retry, as described in [Section 17.2.5](#).

7.3.1. Values of Transport Parameters for 0-RTT

Both endpoints store the value of the server transport parameters from a connection and apply them to any 0-RTT packets that are sent in subsequent connections to that peer, except for transport parameters that are explicitly excluded. Remembered transport parameters apply to the new connection until the handshake completes and the client starts sending 1-RTT packets. Once the handshake completes, the client uses the transport parameters established in the handshake.

The definition of new transport parameters ([Section 7.3.2](#)) **MUST** specify whether they **MUST**, **MAY**, or **MUST NOT** be stored for 0-RTT. A client need not store a transport parameter it cannot process.

A client **MUST NOT** use remembered values for the following parameters: `original_connection_id`, `preferred_address`, `stateless_reset_token`,

`ack_delay_exponent` and `active_connection_id_limit`. The client MUST use the server's new values in the handshake instead, and absent new values from the server, the default value.

A client that attempts to send 0-RTT data MUST remember all other transport parameters used by the server. The server can remember these transport parameters, 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.

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 the following parameters ([Section 18.2](#)) that are smaller than the remembered value of the parameters.

- o `initial_max_data`
- o `initial_max_stream_data_bidi_local`
- o `initial_max_stream_data_bidi_remote`
- o `initial_max_stream_data_uni`
- o `initial_max_streams_bidi`
- o `initial_max_streams_uni`

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_streams_bidi` and `initial_max_stream_data_bidi_remote`, or `initial_max_streams_uni` and `initial_max_stream_data_uni`.

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

When sending frames in 0-RTT packets, a client MUST only use remembered transport parameters; importantly, it MUST NOT use updated values that it learns from the server's updated transport parameters or from frames received in 1-RTT packets. Updated values of transport parameters from the handshake apply only to 1-RTT packets. For instance, flow control limits from remembered transport parameters apply to all 0-RTT packets even if those values are increased by the handshake or by frames sent in 1-RTT packets. A

server MAY treat use of updated transport parameters in 0-RTT as a connection error of type `PROTOCOL_VIOLATION`.

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. As described in [Section 18.1](#), some identifiers are reserved in order to exercise this requirement.

New transport parameters can be registered according to the rules in [Section 22.1](#).

7.4. Cryptographic Message Buffering

Implementations need to maintain a buffer of CRYPTO data received out of order. Because there is no flow control of CRYPTO frames, an endpoint could potentially force its peer to buffer an unbounded amount of data.

Implementations MUST support buffering at least 4096 bytes of data received in CRYPTO frames out of order. Endpoints MAY choose to allow more data to be buffered during the handshake. A larger limit during the handshake could allow for larger keys or credentials to be exchanged. An endpoint's buffer size does not need to remain constant during the life of the connection.

Being unable to buffer CRYPTO frames during the handshake can lead to a connection failure. If an endpoint's buffer is exceeded during the handshake, it can expand its buffer temporarily to complete the handshake. If an endpoint does not expand its buffer, it MUST close the connection with a `CRYPTO_BUFFER_EXCEEDED` error code.

Once the handshake completes, if an endpoint is unable to buffer all data in a CRYPTO frame, it MAY discard that CRYPTO frame and all CRYPTO frames received in the future, or it MAY close the connection with a `CRYPTO_BUFFER_EXCEEDED` error code. Packets containing discarded CRYPTO frames MUST be acknowledged because the packet has been received and processed by the transport even though the CRYPTO frame was discarded.

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. In determining this limit, servers only count the size of successfully processed packets.

Clients **MUST** ensure that UDP datagrams containing only Initial packets are sized to at least 1200 bytes, adding padding to packets in the datagram as necessary. Sending padded datagrams ensures that the server is not overly constrained by the amplification restriction.

Packet loss, in particular loss of a Handshake packet from the server, can cause a situation in which the server cannot send when the client has no data to send and the anti-amplification limit is reached. In order to avoid this causing a handshake deadlock, clients **SHOULD** send a packet upon a crypto retransmission 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 bytes. 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. QUIC uses a token in the Initial packet to provide address validation prior to completing the handshake. This token is delivered to the client 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](#)).

In addition to sending limits imposed prior to address validation, servers are also constrained in what they can send by the limits set by the congestion controller. Clients are only constrained by the congestion controller.

8.1.1. Address Validation using Retry Packets

Upon receiving the client's Initial packet, the server can request address validation by sending a Retry packet ([Section 17.2.5](#)) containing a token. This token MUST be repeated by the client in all Initial packets it sends for that connection after it receives the Retry packet. In response to processing an Initial containing a token, a server can either abort the connection or permit it to proceed.

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.

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 5.

| Client | Server | |
|---------------------------------|---|--|
| Initial[0]: CRYPTO[CH] -> | | |
| | <- Retry+Token | |
| Initial+Token[1]: CRYPTO[CH] -> | | |
| | Initial[0]: CRYPTO[SH] ACK[1] | |
| | Handshake[0]: CRYPTO[EE, CERT, CV, FIN] | |
| | <- 1-RTT[0]: STREAM[1, "..."] | |

Figure 5: 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 19.7](#) to provide the client with an address validation token that can be used to validate future connections. The client includes this token in Initial packets to provide address validation in a future connection. The client MUST include the token in all Initial packets it sends, unless a Retry replaces the token with a newer one. The client MUST NOT use the token provided in a Retry for future connections. Servers MAY discard any Initial packet that does not carry the expected token.

A token SHOULD be constructed in a way that allows the server to distinguish it from tokens that are sent in Retry packets as they are carried in the same field.

The token MUST NOT include information that would allow it to be linked by an on-path observer to the connection on which it was issued. For example, it cannot include the connection ID or addressing information unless the values are encrypted.

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 token SHOULD have an expiration time, which could be either an explicit expiration time or an issued timestamp that can be used to dynamically calculate the expiration time. A server can store the expiration time or include it in an encrypted form in the token.

It is unlikely that the client port number is the same on two different connections; validating the port is therefore unlikely to be successful.

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 SHOULD include that value in the Token field of its Initial packet. Including a token might allow the server to validate the client address without an additional round trip.

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. A token obtained in a Retry packet MUST be used immediately during the connection attempt and cannot be used in subsequent connection attempts.

A client SHOULD NOT reuse a token in different connections. 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 there is any change in its local address prior to completing the handshake.

Clients might receive multiple tokens on a single connection. Aside from preventing linkability, any token can be used in any connection attempt. Servers can send additional tokens to either enable address validation for multiple connection attempts or to replace older tokens that might become invalid. For a client, this ambiguity means that sending the most recent unused token is most likely to be effective. Though saving and using older tokens has no negative consequences, clients can regard older tokens as being less likely to be useful to the server for address validation.

When a server receives an Initial packet with an address validation token, it MUST attempt to validate the token, unless it has already completed address validation. 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 SHOULD 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 to validate client addresses.

Attackers could replay tokens to use servers as amplifiers in DDoS attacks. To protect against such attacks, servers SHOULD ensure that tokens sent in Retry packets are only accepted for a short time. Tokens that are provided in NEW_TOKEN frames (see [Section 19.7](#)) need to be valid for longer, but SHOULD NOT be accepted multiple times in

a short period. Servers are encouraged to allow tokens to be used only once, if possible.

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 (PATH_CHALLENGE) can be both sent to and received (PATH_RESPONSE) 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, however an endpoint SHOULD NOT send multiple PATH_CHALLENGE frames in a single packet. 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 unpredictable data in every PATH_CHALLENGE frame so that it can associate the peer's response with the corresponding 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.

An endpoint MUST NOT send more than one PATH_RESPONSE frame in response to one PATH_CHALLENGE frame (see [Section 13.3](#)). The peer is expected to send more PATH_CHALLENGE frames as necessary to evoke additional PATH_RESPONSE frames.

8.5. Successful Path Validation

A new address is considered valid when a PATH_RESPONSE frame is received that contains the 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.

Note that receipt on a different local address does not result in path validation failure, as it might be a result of a forwarded packet (see [Section 9.3.3](#)) or misrouting. It is possible that a valid `PATH_RESPONSE` might be received in the future.

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. A value of three times the larger of the current Probe Timeout (PTO) or the initial timeout (that is, $2 * kInitialRtt$) as defined in [\[QUIC-RECOVERY\]](#) is **RECOMMENDED**. That is:

```
validation_timeout = max(3*PTO, 6*kInitialRtt)
```

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 (IP address and 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.

The design of QUIC relies on endpoints retaining a stable address for the duration of the handshake. An endpoint **MUST NOT** initiate connection migration before the handshake is confirmed, as defined in section 4.1.2 of [\[QUIC-TLS\]](#).

An endpoint also **MUST NOT** send packets from a different local address, actively initiating migration, if the peer sent the

"disable_active_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 MUST either drop the incoming packets on that path without generating a stateless reset or proceed with path validation and allow the peer to migrate. Generating a stateless reset or closing the connection would allow third parties in the network to cause connections to close by spoofing or otherwise manipulating observed traffic.

Not all changes of peer address are intentional, or active, 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. An endpoint MUST perform path validation ([Section 8.2](#)) if it detects any change to a peer's address, unless it has previously validated that address.

When an endpoint has no validated path on which to send packets, it MAY discard connection state. An endpoint capable of connection migration MAY wait for a new path to become available before discarding connection state.

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 MUST 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 13.4](#).

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. Peer Address Spoofing

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. On-Path Address Spoofing

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.

9.3.3. Off-Path Packet Forwarding

An off-path attacker that can observe packets might forward copies of genuine packets to endpoints. If the copied packet arrives before the genuine packet, this will appear as a NAT rebinding. Any genuine packet will be discarded as a duplicate. If the attacker is able to continue forwarding packets, it might be able to cause migration to a path via the attacker. This places the attacker on path, giving it the ability to observe or drop all subsequent packets.

Unlike the attack described in [Section 9.3.2](#), the attacker can ensure that the new path is successfully validated.

This style of attack relies on the attacker using a path that is approximately as fast as the direct path between endpoints. The attack is more reliable if relatively few packets are sent or if packet loss coincides with the attempted attack.

A non-probing packet received on the original path that increases the maximum received packet number will cause the endpoint to move back to that path. Eliciting packets on this path increases the likelihood that the attack is unsuccessful. Therefore, mitigation of this attack relies on triggering the exchange of packets.

In response to an apparent migration, endpoints **MUST** validate the previously active path using a `PATH_CHALLENGE` frame. This induces the sending of new packets on that path. If the path is no longer viable, the validation attempt will time out and fail; if the path is viable, but no longer desired, the validation will succeed, but only results in probing packets being sent on the path.

An endpoint that receives a `PATH_CHALLENGE` on an active path **SHOULD** send a non-probing packet in response. If the non-probing packet

arrives before any copy made by an attacker, this results in the connection being migrated back to the original path. Any subsequent migration to another path restarts this entire process.

This defense is imperfect, but this is not considered a serious problem. If the path via the attack is reliably faster than the original path despite multiple attempts to use that original path, it is not possible to distinguish between attack and an improvement in routing.

An endpoint could also use heuristics to improve detection of this style of attack. For instance, NAT rebinding is improbable if packets were recently received on the old path, similarly rebinding is rare on IPv6 paths. Endpoints can also look for duplicated packets. Conversely, a change in connection ID is more likely to indicate an intentional migration rather than an attack.

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 **MUST** immediately reset the congestion controller and round-trip time estimator for the new path to initial values (see Sections A.3 and B.3 in [QUIC-RECOVERY]) unless it has knowledge that a previous send rate or round-trip time estimate is valid for the new path. For instance, an endpoint might infer that a change in only the client's port number is indicative of a NAT rebinding, meaning that the new path is likely to have similar bandwidth and round-trip time. However, this determination will be imperfect. If the determination is incorrect, the congestion controller and the RTT estimator are expected to adapt to the new path. Generally, implementations are advised to be cautious when using previous values on a new path.

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. For instance, an endpoint might delay switching to a new congestion control context until it is confirmed that an old path is no longer needed (such as the case in [Section 9.3.3](#)).

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.

At any time, endpoints MAY change the Destination Connection ID they send to a value that has not been used on another path.

An endpoint MUST use a new connection ID if it initiates connection migration. Using a new connection ID eliminates the use of the connection ID for linking activity from the same connection on different networks. Header protection 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.

Unintentional changes in path without a change in connection ID are possible. For example, after a period of network inactivity, NAT rebinding might cause packets to be sent on a new path when the client resumes sending.

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.

An endpoint that exhausts available connection IDs cannot migrate. To ensure that migration is possible and packets sent on different

paths cannot be correlated, endpoints SHOULD provide new connection IDs before peers migrate.

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.

Servers MAY communicate a preferred address of each address family (IPv4 and IPv6) to allow clients to pick the one most suited to their network attachment.

Once the handshake is finished, the client SHOULD select one of the two server's preferred addresses and initiate path validation (see [Section 8.2](#)) of that 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 MUST send other non-probing frames from its original address until it receives a non-probing packet from the client at its preferred address and until the server has validated the new path.

The server **MUST** probe on the path toward the client from its preferred address. 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.

A client that migrates to a new address **SHOULD** use a preferred address from the same address family for the server.

9.7. Use of IPv6 Flow-Label and Migration

Endpoints that send data using IPv6 **SHOULD** apply an IPv6 flow label in compliance with [\[RFC6437\]](#), unless the local API does not allow setting IPv6 flow labels.

The IPv6 flow label **SHOULD** be a pseudo-random function of the source and destination addresses, source and destination UDP ports, and the

destination CID. The flow label generation MUST be designed to minimize the chances of linkability with a previously used flow label, as this would enable correlating activity on multiple paths (see [Section 9.5](#)).

A possible implementation is to compute the flow label as a cryptographic hash function of the source and destination addresses, source and destination UDP ports, destination CID, and a local secret.

10. Connection Termination

An established QUIC connection can be terminated in one of three ways:

- o idle timeout ([Section 10.2](#))
- o immediate close ([Section 10.3](#))
- o stateless reset ([Section 10.4](#))

An endpoint MAY discard connection state if it does not have a validated path on which it can send packets (see [Section 8.2](#)).

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 at least three times the current Probe Timeout (PTO) 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 frame (see [Section 10.3](#) for details). An endpoint retains only enough information to generate a packet containing a CONNECTION_CLOSE frame and to identify packets as belonging to the connection. The endpoint's selected connection ID and the QUIC version are 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 CONNECTION_CLOSE 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 receives a `CONNECTION_CLOSE` frame or stateless reset, both of which indicate that the peer is also closing or draining. 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 generate an unnecessary stateless reset. Endpoints that have some alternative means to ensure that late-arriving packets on the connection do not induce a response, 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.

While in the closing period, an endpoint could receive packets from a new source address, indicating a connection migration ([Section 9](#)). 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 is silently closed and the state is discarded when it remains idle for longer than both the advertised idle timeout (see [Section 18.2](#)) and three times the current Probe Timeout (PTO).

Each endpoint advertises its own idle timeout to its peer. An endpoint restarts any timer it maintains when a packet from its peer

is received and processed successfully. The timer is also restarted when sending a packet containing frames other than ACK or PADDING (an ACK-eliciting packet; see [QUIC-RECOVERY]), but only if no other ACK-eliciting packets have been sent since last receiving a packet. Restarting when sending packets ensures that connections do not prematurely time out when initiating new activity.

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 a Probe Timeout (PTO; see Section 6.3 of [QUIC-RECOVERY]), it is advisable to test for liveness before sending any data that cannot be retried safely. Note that it is likely that only applications or application protocols will know what information can be retried.

10.3. Immediate Close

An endpoint sends a CONNECTION_CLOSE frame (Section 19.19) to terminate the connection immediately. A CONNECTION_CLOSE frame causes all streams to immediately become closed; open streams can be assumed to be implicitly reset.

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

New packets from unverified addresses could be used to create an amplification attack (see Section 8). To avoid this, endpoints MUST either limit transmission of CONNECTION_CLOSE frames to validated addresses or drop packets without response if the response would be more than three times larger than the received packet.

After receiving a `CONNECTION_CLOSE` frame, endpoints enter the draining state. An endpoint that receives a `CONNECTION_CLOSE` frame MAY send a single packet containing a `CONNECTION_CLOSE` 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 `CONNECTION_CLOSE` 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 a `CONNECTION_CLOSE` frame with an appropriate error code to signal closure.

When sending `CONNECTION_CLOSE`, the goal is to ensure that the peer will process the frame. Generally, this means sending the frame in a packet with the highest level of packet protection to avoid the packet being discarded. However, during the handshake, it is possible that more advanced packet protection keys are not available to the peer, so the frame MAY be replicated in a packet that uses a lower packet protection level.

After the handshake is confirmed, an endpoint MUST send any `CONNECTION_CLOSE` frames in a 1-RTT packet. Prior to handshake confirmation, the peer might not have 1-RTT keys, so the endpoint SHOULD send `CONNECTION_CLOSE` frames in a Handshake packet. If the endpoint does not have Handshake keys, it SHOULD send `CONNECTION_CLOSE` frames in an Initial packet.

A client will always know whether the server has Handshake keys (see [Section 17.2.2.1](#)), but it is possible that a server does not know whether the client has Handshake keys. Under these circumstances, a server SHOULD send a `CONNECTION_CLOSE` frame in both Handshake and Initial packets to ensure that at least one of them is processable by the client. These packets can be coalesced into a single UDP datagram (see [Section 12.2](#)).

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 MAY send a stateless reset in response to receiving a packet that it cannot associate with an active connection.

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 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 are invalidated when their associated connection ID is retired via a RETIRE_CONNECTION_ID frame ([Section 19.16](#)).

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

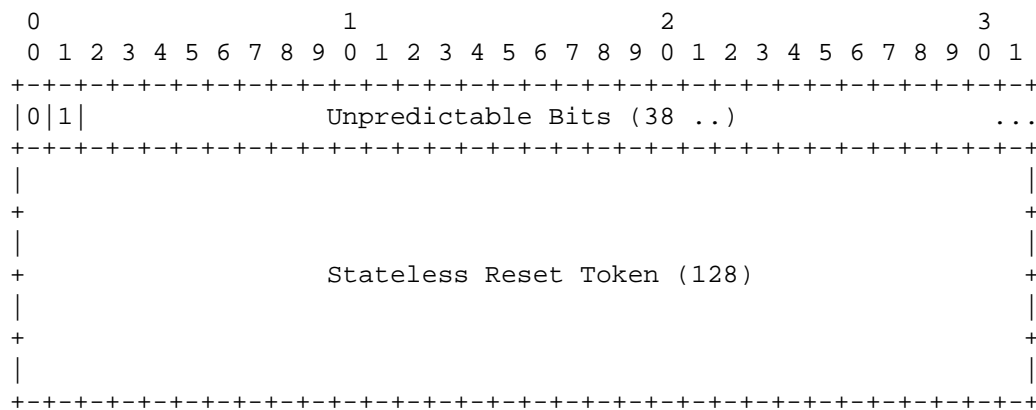


Figure 6: Stateless Reset Packet

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

A stateless reset uses an entire UDP datagram, starting with the first two bits of the packet header. The remainder of the first byte and an arbitrary number of bytes following it that are set to unpredictable values. The last 16 bytes of the datagram contain a Stateless Reset Token.

To entities other than its intended recipient, a stateless reset will appear to be a packet with a short header. For the stateless reset to appear as a valid QUIC packet, the Unpredictable Bits field needs

to include at least 38 bits of data (or 5 bytes, less the two fixed bits).

A minimum size of 21 bytes does not guarantee that a stateless reset is difficult to distinguish from other packets if the recipient requires the use of a connection ID. To prevent a resulting stateless reset from being trivially distinguishable from a valid packet, all packets sent by an endpoint SHOULD be padded to at least 22 bytes longer than the minimum connection ID that the endpoint might use. An endpoint that sends a stateless reset in response to packet that is 43 bytes or less in length SHOULD send a stateless reset that is one byte shorter than the packet it responds to.

These values assume that the Stateless Reset Token is the same as the minimum expansion of the packet protection AEAD. Additional unpredictable bytes are necessary if the endpoint could have negotiated a packet protection scheme with a larger minimum expansion.

An endpoint MUST NOT send a stateless reset that is three times or more larger than the packet it receives to avoid being used for amplification. [Section 10.4.3](#) describes additional limits on stateless reset size.

Endpoints MUST discard packets that are too small to be valid QUIC packets. With the set of AEAD functions defined in [\[QUIC-TLS\]](#), packets that are smaller than 21 bytes are never valid.

Endpoints MUST send stateless reset packets formatted as a packet with a short header. However, endpoints MUST treat any packet ending in a valid stateless reset token as a stateless reset, as other QUIC versions might allow the use of a long header.

An endpoint MAY send a stateless reset in response to a packet with a long header. Sending a stateless reset is not effective prior to the stateless reset token being 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 as effective as sending a stateless reset.

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 19.15](#)).

Using a randomized connection ID results in two problems:

- o 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 an ineffective error detection and recovery mechanism. In this case, endpoints will need to rely on other methods - such as timers - to detect that the connection has failed.
- o 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.

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 bytes for carrying data.

[10.4.1](#). Detecting a Stateless Reset

An endpoint detects a potential stateless reset when an incoming packet either cannot be associated with a connection, cannot be decrypted, or is marked as a duplicate packet. The endpoint MUST then compare the last 16 bytes of the packet with all Stateless Reset Tokens that are associated with connection IDs that the endpoint recently used to send packets from the IP address and port on which the datagram is received. This includes Stateless Reset Tokens from NEW_CONNECTION_ID frames and the server's transport parameters. An endpoint MUST NOT check for any Stateless Reset Tokens associated with connection IDs it has not used or for connection IDs that have been retired.

If the last 16 bytes of the packet values are identical to a Stateless Reset Token, 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 bytes 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 MUST NOT be used 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 21.9](#). A connection ID from a connection that is reset by revealing the Stateless Reset Token MUST NOT be reused for new connections at nodes that share a static key.

The same Stateless Reset Token MAY be used for multiple connection IDs on the same connection. However, reuse of a Stateless Reset Token might expose an endpoint to denial of service if associated connection IDs are forgotten while the associated token is still active at a peer. An endpoint MUST ensure that packets with

Destination Connection ID field values that correspond to a reused Stateless Reset Token are attributed to the same connection as long as the Stateless Reset Token is still usable, even when the connection ID has been retired. Otherwise, an attacker might be able to send a packet with a retired connection ID and cause the endpoint to produce a Stateless Reset that it can use to disrupt the connection, just as with the attacks in [Section 21.9](#).

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

10.4.3. Looping

The design of a Stateless Reset is such that without knowing the stateless reset token it is indistinguishable from a valid packet. For instance, if a server sends a Stateless Reset to another server it might receive another Stateless Reset in response, which could lead to an infinite exchange.

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 41 bytes means that the packet could reveal to an observer that it is a Stateless Reset, depending upon the length of the peer's connection IDs. 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.

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; though these are worded as requirements, different implementation strategies might lead to different errors being reported. In particular, an endpoint MAY use any applicable error code when it detects an error condition; a generic error code (such as `PROTOCOL_VIOLATION` or `INTERNAL_ERROR`) can always be used in place of specific error codes.

A stateless reset ([Section 10.4](#)) is not suitable for any error that can be signaled with a `CONNECTION_CLOSE` or `RESET_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` frame ([Section 19.19](#)). 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-specific variant of the `CONNECTION_CLOSE` frame. Errors that are specific to the transport, including all those described in this document, are carried in the QUIC-specific variant of the `CONNECTION_CLOSE` frame.

A `CONNECTION_CLOSE` frame could be sent in a packet that is lost. An endpoint SHOULD be prepared to retransmit a packet containing a `CONNECTION_CLOSE` frame 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 a `CONNECTION_CLOSE` frame 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` 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 RESET_STREAM frame ([Section 19.4](#)) with an appropriate error code to terminate just the affected stream.

RESET_STREAM MUST be instigated by the protocol using QUIC. RESET_STREAM carries an application error code. Only the application protocol is able to cause a stream to be terminated. A local instance of the application protocol uses a direct API call and a remote instance uses the STOP_SENDING frame, which triggers an automatic RESET_STREAM.

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. Application protocols SHOULD define rules for handling streams that are prematurely cancelled by either endpoint.

12. Packets and Frames

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

This version of QUIC uses the long packet header (see [Section 17.2](#)) during connection establishment. Packets with the long header are Initial ([Section 17.2.2](#)), 0-RTT ([Section 17.2.3](#)), Handshake ([Section 17.2.4](#)), and Retry ([Section 17.2.5](#)). Version negotiation uses a version-independent packet with a long header (see [Section 17.2.1](#)).

Packets with the short header ([Section 17.3](#)) are designed for minimal overhead and are used after a connection is established and 1-RTT keys are available.

12.1. Protected Packets

All QUIC packets except Version Negotiation and Retry packets use authenticated encryption with additional data (AEAD) [[RFC5116](#)] 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.

Initial protection 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 in a given packet number space; see [Section 12.3](#) for details.

12.2. Coalescing Packets

Initial ([Section 17.2.2](#)), 0-RTT ([Section 17.2.3](#)), and Handshake ([Section 17.2.4](#)) packets contain a Length field, which determines the end of the packet. The length includes both the Packet Number and Payload fields, both of which are confidentiality protected and initially of unknown length. The length of the Payload field is learned once header protection is removed.

Using the Length field, 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 start sending data. This can also be used to construct PMTU probes (see [Section 14.3.1](#)). 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 can only be the last packet included in a UDP datagram. An endpoint SHOULD NOT coalesce multiple packets at the same encryption level.

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), the receiver MAY either discard or buffer the packet for later processing and MUST attempt to process the remaining packets.

Retry packets ([Section 17.2.5](#)), Version Negotiation packets ([Section 17.2.1](#)), and packets with a short header ([Section 17.3](#)) do not contain a Length field and so cannot be followed by other packets in the same UDP datagram. Note also that there is no situation where a Retry or Version Negotiation packet is coalesced with another packet.

12.3. Packet Numbers

The packet number is an integer in the range 0 to $2^{62}-1$. This number is used in determining the cryptographic nonce for packet protection. Each endpoint maintains a separate packet number for sending and receiving.

Packet numbers are limited to this range because they need to be representable in whole in the Largest Acknowledged field of an ACK frame ([Section 19.3](#)). When present in a long or short header however, packet numbers are reduced and encoded in 1 to 4 bytes (see [Section 17.1](#)).

Version Negotiation ([Section 17.2.1](#)) and Retry ([Section 17.2.5](#)) packets do not include a packet number.

Packet numbers are divided into 3 spaces in QUIC:

- o Initial space: All Initial packets ([Section 17.2.2](#)) are in this space.
- o Handshake space: All Handshake packets ([Section 17.2.4](#)) are in this space.
- o 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. Packet numbers in each space start 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. 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 [[RFC4303](#)].

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, consists of a sequence of complete frames, as shown in [Figure 7](#). Version Negotiation, Stateless Reset, and Retry packets do not contain frames.

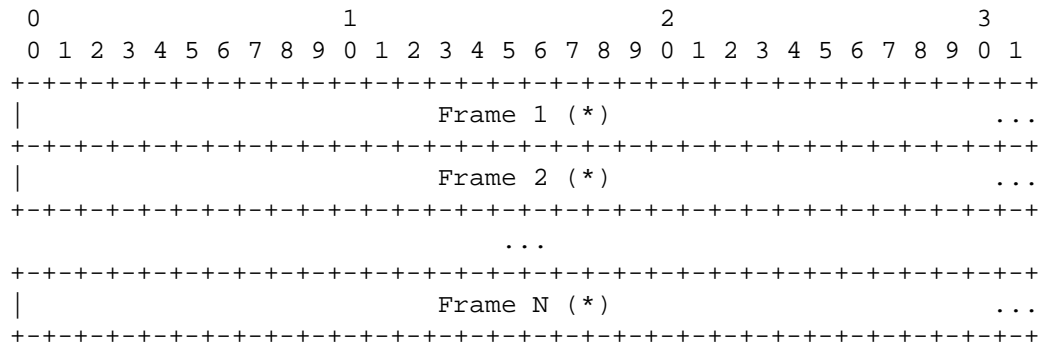


Figure 7: QUIC Payload

The payload of a packet that contains frames **MUST** contain at least one frame, and **MAY** contain multiple frames and multiple frame types. Frames always fit within a single QUIC packet and cannot span multiple packets.

Each frame begins with a Frame Type, indicating its type, followed by additional type-dependent fields:

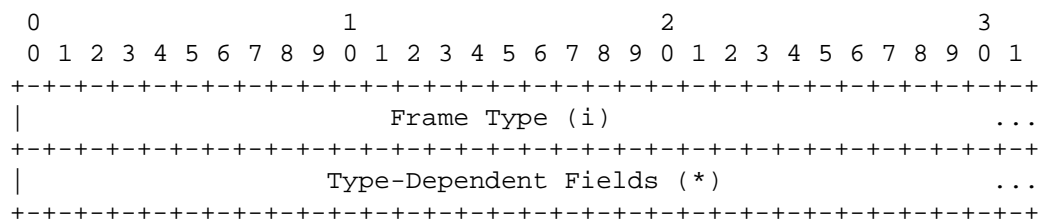


Figure 8: Generic Frame Layout

The frame types defined in this specification are listed in Table 3. The Frame Type in ACK, STREAM, MAX_STREAMS, STREAMS_BLOCKED, and CONNECTION_CLOSE 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 | PING | Section 19.2 |
| 0x02 - 0x03 | ACK | Section 19.3 |
| 0x04 | RESET_STREAM | Section 19.4 |
| 0x05 | STOP_SENDING | Section 19.5 |
| 0x06 | CRYPTO | Section 19.6 |
| 0x07 | NEW_TOKEN | Section 19.7 |
| 0x08 - 0x0f | STREAM | Section 19.8 |
| 0x10 | MAX_DATA | Section 19.9 |
| 0x11 | MAX_STREAM_DATA | Section 19.10 |
| 0x12 - 0x13 | MAX_STREAMS | Section 19.11 |
| 0x14 | DATA_BLOCKED | Section 19.12 |
| 0x15 | STREAM_DATA_BLOCKED | Section 19.13 |
| 0x16 - 0x17 | STREAMS_BLOCKED | Section 19.14 |
| 0x18 | NEW_CONNECTION_ID | Section 19.15 |
| 0x19 | RETIRE_CONNECTION_ID | Section 19.16 |
| 0x1a | PATH_CHALLENGE | Section 19.17 |
| 0x1b | PATH_RESPONSE | Section 19.18 |
| 0x1c - 0x1d | CONNECTION_CLOSE | Section 19.19 |

Table 3: Frame Types

An endpoint MUST treat the receipt of a frame of unknown type as a connection error of type FRAME_ENCODING_ERROR.

All QUIC frames are idempotent in this version of QUIC. 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-byte encoding of the frame types defined in this document is possible, the Frame Type field for these frames is encoded on a single byte. For instance, though 0x4001 is a legitimate two-byte encoding for a variable-length integer with a value of 1, PING frames are always encoded as a single byte with the value 0x01. An endpoint **MAY** 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

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.

13.2. Generating Acknowledgements

Endpoints acknowledge all packets they receive and process. However, only ack-eliciting packets (see [QUIC-RECOVERY]) trigger the sending of an ACK frame. Packets that are not ack-eliciting are only acknowledged when an ACK frame is sent for other reasons.

When sending a packet for any reason, an endpoint should attempt to bundle an ACK frame if one has not been sent recently. Doing so helps with timely loss detection at the peer.

In general, frequent feedback from a receiver improves loss and congestion response, but this has to be balanced against excessive load generated by a receiver that sends an ACK frame in response to every ack-eliciting packet. The guidance offered below seeks to strike this balance.

13.2.1. Sending ACK Frames

An ACK frame **SHOULD** be generated for at least every second ack-eliciting packet. This recommendation is in keeping with standard practice for TCP [RFC5681].

A 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.

In order to assist loss detection at the sender, an endpoint **SHOULD** send an ACK frame immediately on receiving an ack-eliciting packet that is out of order. The endpoint **MAY** continue sending ACK frames immediately on each subsequently received packet, but the endpoint **SHOULD** return to acknowledging every other packet after a period of $1/8 \times \text{RTT}$, unless more ACK-eliciting packets are received out of

order. If every subsequent ACK-eliciting packet arrives out of order, then an ACK frame SHOULD be sent immediately for every received ACK-eliciting packet.

Similarly, packets marked with the ECN Congestion Experienced (CE) codepoint in the IP header SHOULD be acknowledged immediately, to reduce the peer's response time to congestion events.

As an optimization, a receiver MAY process multiple packets before sending any ACK frames in response. In this case the receiver can determine whether an immediate or delayed acknowledgement should be generated after processing incoming packets.

Acknowledgements of packets carrying CRYPTO frames SHOULD be minimally delayed, to complete the handshake with minimal latency. Delaying them by a small amount, such as the local timer granularity, allows the endpoint to bundle any data sent in response with the ACK frame. ACK frames SHOULD be sent immediately when the crypto stack indicates all data for that packet number space has been received.

Packets containing PADDING frames 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 that is only sending ACK frames will not receive acknowledgments from its peer unless those acknowledgements are included in packets with ACK-eliciting frames. An endpoint SHOULD bundle ACK frames with other frames when there are new ACK-eliciting packets to acknowledge. When only non-ACK-eliciting packets need to be acknowledged, an endpoint MAY wait until an ACK-eliciting packet has been received to bundle an ACK frame with outgoing frames.

The algorithms in [QUIC-RECOVERY] are resilient to receivers that do not follow guidance offered above. However, an implementor should only deviate from these requirements after careful consideration of the performance implications of doing so.

Packets containing only ACK frames are not congestion controlled, so there are limits on how frequently they can be sent. An endpoint MUST NOT send more than one ACK-frame-only packet in response to receiving an ACK-eliciting packet (one containing frames other than ACK and/or PADDING). An endpoint MUST NOT send a packet containing only an ACK frame in response to a non-ACK-eliciting packet (one containing only ACK and/or PADDING frames), even if there are packet

gaps which precede the received packet. Limiting ACK frames avoids an infinite feedback loop of acknowledgements, which could prevent the connection from ever becoming idle. However, the endpoint acknowledges non-ACK-eliciting packets when it sends an ACK frame.

An endpoint SHOULD treat receipt of an acknowledgment for a packet it did not send as a connection error of type `PROTOCOL_VIOLATION`, if it is able to detect the condition.

13.2.2. Managing ACK Ranges

When an ACK frame is sent, one or more ranges of acknowledged packets are included. Including older packets reduces the chance of spurious retransmits caused by losing previously sent ACK frames, at the cost of larger ACK frames.

ACK frames SHOULD always acknowledge the most recently received packets, and the more out-of-order the packets are, the more important it is to send an updated ACK frame quickly, to prevent the peer from declaring a packet as lost and spuriously retransmitting the frames it contains.

[Section 13.2.3](#) and [Section 13.2.4](#) describe an exemplary approach for determining what packets to acknowledge in each ACK frame.

13.2.3. Receiver Tracking of ACK Frames

When a packet containing an ACK frame is sent, the largest acknowledged in that frame may be saved. When a packet containing an ACK frame is acknowledged, the receiver can stop acknowledging packets less than or equal to the largest acknowledged in the sent ACK frame.

In cases without ACK frame loss, this algorithm allows for a minimum of 1 RTT of reordering. In cases with ACK frame loss and reordering, this approach does not guarantee that every acknowledgement is seen by the sender before it is no longer included in the ACK frame. Packets could be received out of order and all subsequent ACK frames containing them could be lost. In this case, the loss recovery algorithm could cause spurious retransmits, but the sender will continue making forward progress.

13.2.4. Limiting ACK Ranges

To limit ACK Ranges (see [Section 19.3.1](#)) to those that have not yet been received by the sender, the receiver SHOULD track which ACK frames have been acknowledged by its peer. The receiver SHOULD exclude already acknowledged packets from future ACK frames whenever

these packets would unnecessarily contribute to the ACK frame size. When the receiver is only sending non-ACK-eliciting packets, it can bundle a PING or other small ACK-eliciting frame with a fraction of them, such as once per round trip, to enable dropping unnecessary ACK ranges and any state for previously sent packets. The receiver **MUST NOT** bundle an ACK-eliciting frame, such as a PING, with all packets that would otherwise be non-ACK-eliciting, in order to avoid an infinite feedback loop of acknowledgements.

To limit receiver state or the size of ACK frames, a receiver **MAY** limit the number of ACK Ranges 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 algorithms ([[QUIC-RECOVERY](#)]) 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.

13.2.5. Measuring and Reporting Host Delay

An endpoint measures the delays intentionally introduced between when an ACK-eliciting packet is received and the corresponding acknowledgment is sent. The endpoint encodes this delay for the largest acknowledged packet in the Ack Delay field of an ACK frame (see [Section 19.3](#)). This allows the receiver of the ACK to adjust for any intentional delays, which is important for getting a better estimate of the path RTT when acknowledgments are delayed. A packet might be held in the OS kernel or elsewhere on the host before being processed. An endpoint **MUST NOT** include delays that it does not control when populating the Ack Delay field in an ACK frame.

An endpoint **MUST NOT** excessively delay acknowledgements of ack-eliciting packets. An endpoint commits to a maximum delay using the `max_ack_delay` transport parameter; see [Section 18.2](#). `max_ack_delay` declares an explicit contract: an endpoint promises to never delay acknowledgments of an ack-eliciting packet by more than the indicated value. If it does, any excess accrues to the RTT estimate and could result in delayed retransmissions from the peer. For Initial and Handshake packets, a `max_ack_delay` of 0 is used.

13.2.6. 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.

13.3. 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.

- o Data sent in CRYPTO frames is retransmitted according to the rules in [QUIC-RECOVERY], until all data has been acknowledged. Data in CRYPTO frames for Initial and Handshake packets is discarded when keys for the corresponding encryption level are discarded.
- o Application data sent in STREAM frames is retransmitted in new STREAM frames unless the endpoint has sent a RESET_STREAM for that stream. Once an endpoint sends a RESET_STREAM frame, no further STREAM frames are needed.
- o The most recent set of acknowledgments are sent in ACK frames. An ACK frame SHOULD contain all unacknowledged acknowledgments, as described in Section 13.2.1.
- o Cancellation of stream transmission, as carried in a RESET_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 sending part of the stream). The content of a RESET_STREAM frame MUST NOT change when it is sent again.
- o Similarly, a request to cancel stream transmission, as encoded in a STOP_SENDING frame, is sent until the receiving part of the stream enters either a "Data Recvd" or "Reset Recvd" state; see Section 3.5.

- o Connection close signals, including packets that contain CONNECTION_CLOSE frames, are not sent again when packet loss is detected, but as described in [Section 10](#).
- o 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.
- o 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 receiving part of the stream enters a "Size Known" state.
- o The limit on streams of a given type is sent in MAX_STREAMS frames. Like MAX_DATA, an updated value is sent when a packet containing the most recent MAX_STREAMS 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.
- o Blocked signals are carried in DATA_BLOCKED, STREAM_DATA_BLOCKED, and STREAMS_BLOCKED frames. DATA_BLOCKED frames have connection scope, STREAM_DATA_BLOCKED frames have stream scope, and STREAMS_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.
- o 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.
- o Responses to path validation using PATH_RESPONSE frames are sent just once. The peer is expected to send more PATH_CHALLENGE frames as necessary to evoke additional PATH_RESPONSE frames.
- o 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.

- o NEW_TOKEN frames are retransmitted if the packet containing them is lost. No special support is made for detecting reordered and duplicated NEW_TOKEN frames other than a direct comparison of the frame contents.
- o PING and PADDING frames contain no information, so lost PING or PADDING frames do not require repair.

Endpoints SHOULD prioritize retransmission of data over sending new data, unless priorities specified by the application indicate otherwise (see [Section 2.3](#)).

Even though a sender is encouraged to assemble frames containing up-to-date information every time it sends a packet, it is not forbidden to retransmit copies of frames from lost packets. A receiver MUST accept packets containing an outdated frame, such as a MAX_DATA frame carrying a smaller maximum data than one found in an older packet.

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

13.4. Explicit Congestion Notification

QUIC endpoints can 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 validates the use of ECN on the path, both during connection establishment and when migrating to a new path ([Section 9](#)).

13.4.1. ECN Counts

On receiving a QUIC packet with an ECT or CE codepoint, an ECN-enabled endpoint that can access the ECN codepoints from the enclosing IP packet increases the corresponding ECT(0), ECT(1), or CE count, and includes these counts in subsequent ACK frames (see

[Section 13.2](#) and [Section 19.3](#)). Note that this requires being able to read the ECN codepoints from the enclosing IP packet, which is not possible on all platforms.

A packet detected by a receiver as a duplicate does not affect the receiver's local ECN codepoint counts; see ([Section 21.8](#)) for relevant security concerns.

If an endpoint receives a QUIC packet without an ECT or CE codepoint in the IP packet header, it responds per [Section 13.2](#) with an ACK frame without increasing any ECN counts. If an endpoint does not implement ECN support or does not have access to received ECN codepoints, it does not increase ECN counts.

Coalesced packets (see [Section 12.2](#)) mean that several packets can share the same IP header. The ECN counter for the ECN codepoint received in the associated IP header are incremented once for each QUIC packet, not per enclosing IP packet or UDP datagram.

Each packet number space maintains separate acknowledgement state and separate ECN counts. For example, if one each of an Initial, 0-RTT, Handshake, and 1-RTT QUIC packet are coalesced, the corresponding counts for the Initial and Handshake packet number space will be incremented by one and the counts for the 1-RTT packet number space will be increased by two.

13.4.2. ECN Validation

It is possible for faulty network devices to corrupt or erroneously drop packets with ECN markings. To provide robust connectivity in the presence of such devices, each endpoint independently validates ECN counts and disables ECN if errors are detected.

Endpoints validate ECN for packets sent on each network path independently. An endpoint thus validates ECN on new connection establishment, when switching to a new server preferred address, and on active connection migration to a new path.

Even if an endpoint does not use ECN markings on packets it transmits, the endpoint MUST provide feedback about ECN markings received from the peer if they are accessible. Failing to report ECN counts will cause the peer to disable ECN marking.

13.4.2.1. Sending ECN Markings

To start ECN validation, an endpoint SHOULD do the following when sending packets on a new path to a peer:

- o Set the ECT(0) codepoint in the IP header of early outgoing packets sent on a new path to the peer [RFC8311].
- o If all packets that were sent with the ECT(0) codepoint are eventually deemed lost [QUIC-RECOVERY], validation is deemed to have failed.

To reduce the chances of misinterpreting congestive loss as packets dropped by a faulty network element, an endpoint could set the ECT(0) codepoint in the first ten outgoing packets on a path, or for a period of three RTTs, whichever occurs first.

Implementations MAY experiment with and use other strategies for use of ECN. Other methods of probing paths for ECN support are possible, as are different marking strategies. Implementations can also use the ECT(1) codepoint, as specified in [RFC8311].

13.4.2.2. Receiving ACK Frames

An endpoint that sets ECT(0) or ECT(1) codepoints on packets it transmits MUST use the following steps on receiving an ACK frame to validate ECN.

- o If this ACK frame newly acknowledges a packet that the endpoint sent with either ECT(0) or ECT(1) codepoints set, and if no ECN feedback is present in the ACK frame, validation fails. This step protects against both a network element that zeroes out ECN bits and a peer that is unable to access ECN markings, since the peer could respond without ECN feedback in these cases.
- o For validation to succeed, the total increase in ECT(0), ECT(1), and CE counts MUST be no smaller than the total number of QUIC packets sent with an ECT codepoint that are newly acknowledged in this ACK frame. This step detects any network remarking from ECT(0), ECT(1), or CE codepoints to Not-ECT.
- o Any increase in either ECT(0) or ECT(1) counts, plus any increase in the CE count, MUST be no smaller than the number of packets sent with the corresponding ECT codepoint that are newly acknowledged in this ACK frame. This step detects any erroneous network remarking from ECT(0) to ECT(1) (or vice versa).

Processing ECN counts out of order can result in validation failure. An endpoint SHOULD NOT perform this validation if this ACK frame does not advance the largest packet number acknowledged in this connection.

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 counts to be greater than the number of packets acknowledged in an ACK frame. When this happens, and if validation succeeds, the local reference counts MUST be increased to match the counts in the ACK frame.

13.4.2.3. Validation Outcomes

If validation fails, then the endpoint stops sending ECN markings in subsequent IP packets with the expectation that either the network path or the peer does not support ECN.

Upon successful validation, an endpoint can continue to set ECT codepoints in subsequent packets with the expectation that the path is ECN-capable. Network routing and path elements can change mid-connection however; an endpoint MUST disable ECN if validation fails at any point in the connection.

Even if validation fails, an endpoint MAY revalidate ECN on the same path at any later time in the connection.

14. Packet Size

The QUIC packet size includes the QUIC header and protected payload, but not the UDP or IP header.

Clients MUST ensure they send the first Initial packet in a single IP packet. Similarly, the first Initial packet sent after receiving a Retry packet MUST be sent in a single IP packet.

The payload of a UDP datagram carrying the first Initial packet MUST be expanded to at least 1200 bytes, by adding PADDING frames to the Initial packet and/or by coalescing the Initial packet (see [Section 12.2](#)). 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 datagram containing the first Initial packet from a client MAY exceed 1200 bytes 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 bytes. 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 (PMTU)

The PMTU is the maximum size of the entire IP packet including the IP header, UDP header, and UDP payload. The UDP payload includes the QUIC packet header, protected payload, and any authentication fields. The PMTU can depend upon the current path characteristics. Therefore, the current largest UDP payload an implementation will send is referred to as the QUIC maximum packet size.

QUIC depends on a PMTU of at least 1280 bytes. This is the IPv6 minimum size [\[RFC8200\]](#) and is also supported by most modern IPv4 networks. All QUIC packets (except for PMTU probe packets) **SHOULD** be sized to fit within the maximum packet size to avoid the packet being fragmented or dropped [\[RFC8085\]](#).

An endpoint **SHOULD** use Datagram Packetization Layer PMTU Discovery ([\[DPLPMTUD\]](#)) or implement Path MTU Discovery (PMTUD) [\[RFC1191\]](#) [\[RFC8201\]](#) to determine whether the path to a destination will support a desired message size without fragmentation.

In the absence of these mechanisms, QUIC endpoints **SHOULD NOT** send IP packets larger than 1280 bytes. Assuming the minimum IP header size, this results in a QUIC maximum packet size of 1232 bytes for IPv6 and 1252 bytes for IPv4. A QUIC implementation **MAY** be more conservative in computing the QUIC maximum packet size to allow for unknown tunnel overheads or IP header options/extensions.

Each pair of local and remote addresses could have a different PMTU. QUIC implementations that implement any kind of PMTU discovery therefore **SHOULD** maintain a maximum packet size for each combination of local and remote IP addresses.

If a QUIC endpoint determines that the PMTU between any pair of local and remote IP addresses has fallen below the size needed to support the smallest allowed maximum packet size, it **MUST** immediately cease sending QUIC packets, except for PMTU probe packets, on the affected path. An endpoint **MAY** terminate the connection if an alternative path cannot be found.

14.2. ICMP Packet Too Big Messages

PMTU discovery [RFC1191] [RFC8201] relies on reception of ICMP messages (e.g., IPv6 Packet Too Big messages) that indicate when a packet is dropped because it is larger than the local router MTU. DPLPMTUD can also optionally use these messages. This use of ICMP messages is potentially vulnerable to off-path attacks that successfully guess the addresses used on the path and reduce the PMTU to a bandwidth-inefficient value.

An endpoint MUST ignore an ICMP message that claims the PMTU has decreased below 1280 bytes.

The requirements for generating ICMP ([RFC1812], [RFC4443]) state that the quoted packet should contain as much of the original packet as possible without exceeding the minimum MTU for the IP version. The size of the quoted packet can actually be smaller, or the information unintelligible, as described in Section 1.1 of [DPLPMTUD].

QUIC endpoints SHOULD validate ICMP messages to protect from off-path injection as specified in [RFC8201] and Section 5.2 of [RFC8085]. This validation SHOULD use the quoted packet supplied in the payload of an ICMP message to associate the message with a corresponding transport connection [DPLPMTUD].

ICMP message validation MUST include matching IP addresses and UDP ports [RFC8085] and, when possible, connection IDs to an active QUIC session.

Further validation can also be provided:

- o An IPv4 endpoint could set the 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.
- o An endpoint could store additional information from the IP or UDP headers to use for validation (for example, the IP ID or UDP checksum).

The endpoint SHOULD ignore all ICMP messages that fail validation.

An endpoint MUST NOT increase PMTU based on ICMP messages. Any reduction in the QUIC maximum packet size MAY be provisional until QUIC's loss detection algorithm determines that the quoted packet has actually been lost.

14.3. Datagram Packetization Layer PMTU Discovery

Section 6.4 of [DPLPMTUD] provides considerations for implementing Datagram Packetization Layer PMTUD (DPLPMTUD) with QUIC.

When implementing the algorithm in Section 5.3 of [DPLPMTUD], the initial value of BASE_PMTU SHOULD be consistent with the minimum QUIC packet size (1232 bytes for IPv6 and 1252 bytes for IPv4).

PING and PADDING frames can be used to generate PMTU probe packets. These frames might not be retransmitted if a probe packet containing them is lost. However, these frames do consume congestion window, which could delay the transmission of subsequent application data.

A PING frame can be included in a PMTU probe to ensure that a valid probe is acknowledged.

The considerations for processing ICMP messages in the previous section also apply if these messages are used by DPLPMTUD.

14.3.1. PMTU Probes Containing Source Connection ID

Endpoints that rely on the destination connection ID for routing QUIC packets are likely to require that the connection ID be included in PMTU probe packets to route any resulting ICMP messages (Section 14.2) back to the correct endpoint. However, only long header packets (Section 17.2) contain source connection IDs, and long header packets are not decrypted or acknowledged by the peer once the handshake is complete. One way to construct a PMTU probe is to coalesce (see Section 12.2) a Handshake packet (Section 17.2.4) with a short header packet in a single UDP datagram. If the UDP datagram reaches the endpoint, the Handshake packet will be ignored, but the short header packet will be acknowledged. If the UDP datagram elicits an ICMP message, that message will likely contain the source connection ID within the quoted portion of the UDP datagram.

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 are reserved for use in forcing version negotiation to be exercised. That is, any version number where the low four bits of all bytes 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 bytes to encode.

The QUIC variable-length integer encoding reserves the two most significant bits of the first byte to encode the base 2 logarithm of the integer encoding length in bytes. 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 bytes 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 byte sequence c2 19 7c 5e ff 14 e8 8c (in hexadecimal) decodes to the decimal value 151288809941952652; the four byte sequence 9d 7f 3e 7d decodes to 494878333; the two byte sequence 7b bd decodes to 15293; and the single byte 25 decodes to 37 (as does the two byte 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 are integers in the range 0 to $2^{62}-1$ ([Section 12.3](#)). When present in long or short packet headers, they are encoded in 1 to 4 bytes. The number of bits required to represent the packet number is reduced by including the least significant bits of the packet number.

The encoded packet number is protected as described in Section 5.4 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 bit 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 0xab8bc, sending a packet with a number of 0xac5c02 requires a packet number encoding with 16 bits or more; whereas the 24-bit packet number encoding is needed to send a packet with a number of 0xace8fe.

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 header 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 0xa82f30ea, then a packet containing a 16-bit value of 0x9b32 will be decoded as 0xa82f9b32. Example pseudo-code for packet number decoding can be found in [Appendix A](#).

17.2. Long Header Packets

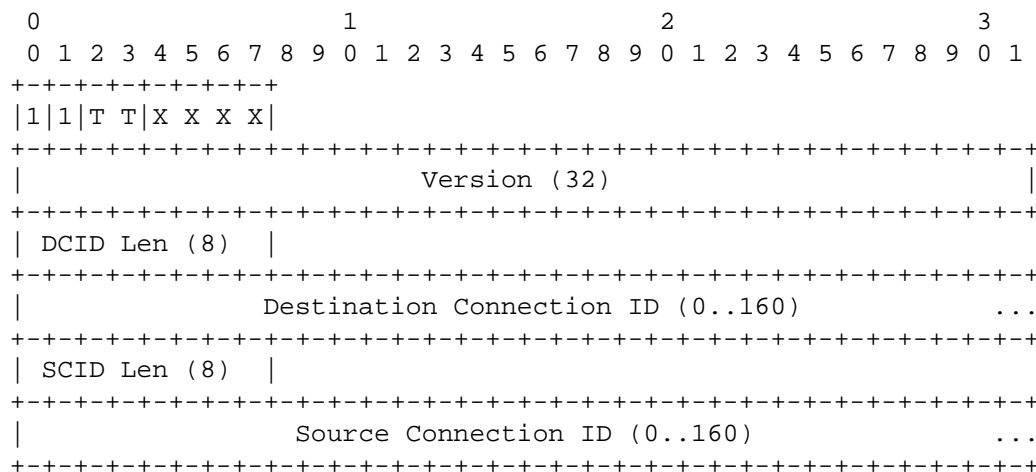


Figure 9: Long Header Packet Format

Long headers are used for packets that are sent prior to the 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 byte 0 (the first byte) is set to 1 for long headers.

Fixed Bit: The next bit (0x40) of byte 0 is set to 1. Packets containing a zero value for this bit are not valid packets in this version and MUST be discarded.

Long Packet Type (T): The next two bits (those with a mask of 0x30) of byte 0 contain a packet type. Packet types are listed in Table 5.

Type-Specific Bits (X): The lower four bits (those with a mask of 0x0f) of byte 0 are type-specific.

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

DCID Len: The byte following the version contains the length in bytes of the Destination Connection ID field that follows it. This length is encoded as an 8-bit unsigned integer. In QUIC version 1, this value MUST NOT exceed 20. Endpoints that receive a version 1 long header with a value larger than 20 MUST drop the packet. Servers SHOULD be able to read longer connection IDs from other QUIC versions in order to properly form a version negotiation packet.

Destination Connection ID: The Destination Connection ID field follows the DCID Len and is between 0 and 20 bytes in length. [Section 7.2](#) describes the use of this field in more detail.

SCID Len: The byte following the Destination Connection ID contains the length in bytes of the Source Connection ID field that follows it. This length is encoded as a 8-bit unsigned integer. In QUIC version 1, this value MUST NOT exceed 20 bytes. Endpoints that receive a version 1 long header with a value larger than 20 MUST drop the packet. Servers SHOULD be able to read longer connection IDs from other QUIC versions in order to properly form a version negotiation packet.

Source Connection ID: The Source Connection ID field follows the SCID Len and is between 0 and 20 bytes in length. [Section 7.2](#) describes the use of this field in more detail.

In this version of QUIC, the following packet types with the long header are defined:

| Type | Name | Section |
|------|-----------|--------------------------------|
| 0x0 | Initial | Section 17.2.2 |
| 0x1 | 0-RTT | Section 17.2.3 |
| 0x2 | Handshake | Section 17.2.4 |
| 0x3 | Retry | Section 17.2.5 |

Table 5: Long Header Packet Types

The header form bit, connection ID lengths byte, Destination and Source Connection ID fields, and Version fields of a long header packet are version-independent. The other fields in the first byte 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. While type-specific semantics for this version are described in the following sections, several long-header packets in this version of QUIC contain these additional fields:

Reserved Bits (R): Two bits (those with a mask of 0x0c) of byte 0 are reserved across multiple packet types. These bits are protected using header protection (see Section 5.4 of [\[QUIC-TLS\]](#)). The value included prior to protection MUST be set to 0. An endpoint MUST treat receipt of a packet that has a non-zero value for these bits, after removing both packet and header protection, as a connection error of type `PROTOCOL_VIOLATION`. Discarding such a packet after only removing header protection can expose the endpoint to attacks (see Section 9.3 of [\[QUIC-TLS\]](#)).

Packet Number Length (P): In packet types which contain a Packet Number field, the least significant two bits (those with a mask of 0x03) of byte 0 contain the length of the packet number, encoded as an unsigned, two-bit integer that is one less than the length of the packet number field in bytes. That is, the length of the packet number field is the value of this field, plus one. These

bits are protected using header protection (see Section 5.4 of [QUIC-TLS]).

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

Packet Number: The packet number field is 1 to 4 bytes long. The packet number has confidentiality protection separate from packet protection, as described in Section 5.4 of [QUIC-TLS]. The length of the packet number field is encoded in the Packet Number Length bits of byte 0 (see above).

17.2.1. Version Negotiation Packet

A Version Negotiation packet is inherently not version-specific. Upon receipt by a client, it 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:

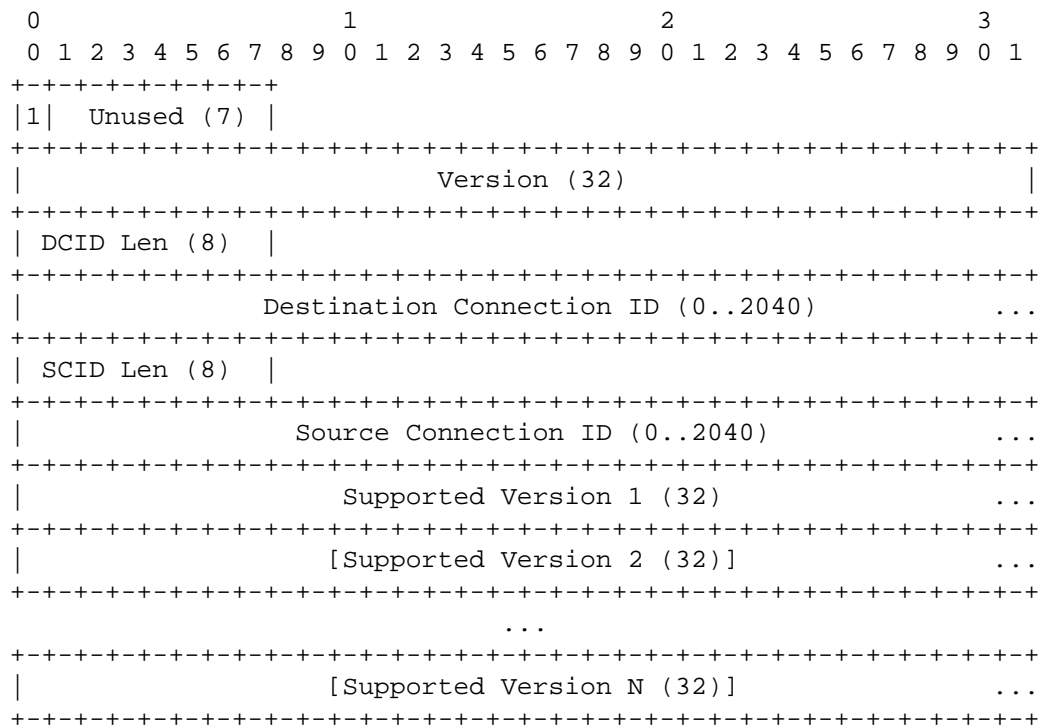


Figure 10: Version Negotiation Packet

The value in the Unused field is selected randomly by the server. Clients MUST ignore the value of this field. Servers SHOULD set the most significant bit of this field (0x40) to 1 so that Version Negotiation packets appear to have the Fixed Bit field.

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.

As future versions of QUIC may support Connection IDs larger than the version 1 limit, Version Negotiation packets could carry Connection IDs that are longer than 20 bytes.

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.

A server **MUST NOT** send more than one Version Negotiation packet in response to a single UDP datagram.

See [Section 6](#) for a description of the version negotiation process.

17.2.2. Initial Packet

An Initial packet uses long headers with a type value of 0x0. It carries the first CRYPTO frames sent by the client and server to perform key exchange, and carries ACKs in either direction.

```

+-----+
| 1 | 1 | 0 | R | R | P | P |
+-----+
|                                     Version (32)                                     |
+-----+
| DCID Len (8) |
+-----+
|                                     Destination Connection ID (0..160)                                     ...
+-----+
| SCID Len (8) |
+-----+
|                                     Source Connection ID (0..160)                                     ...
+-----+
|                                     Token Length (i)                                     ...
+-----+
|                                     Token (*)                                     ...
+-----+
|                                     Length (i)                                     ...
+-----+
|                                     Packet Number (8/16/24/32)                                     ...
+-----+
|                                     Payload (*)                                     ...
+-----+

```

Figure 11: Initial Packet

The Initial packet contains a long header as well as the Length and Packet Number fields. The first byte contains the Reserved and Packet Number Length bits. Between the SCID and Length fields, there are two additional field specific to the Initial packet.

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 that was previously provided in a Retry packet or `NEW_TOKEN` frame.

Payload: The payload of the packet.

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.

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 Retry packet ([Section 17.2.5](#)).

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.

17.2.2.1. Abandoning Initial Packets

A client stops both sending and processing Initial packets when it sends its first Handshake packet. A server stops sending and processing Initial packets when it receives its first Handshake packet. Though packets might still be in flight or awaiting acknowledgment, no further Initial packets need to be exchanged beyond this point. Initial packet protection keys are discarded (see Section 4.9.1 of [QUIC-TLS]) along with any loss recovery and congestion control state (see Section 6.5 of [QUIC-RECOVERY]).

Any data in CRYPTO frames is discarded - and no longer retransmitted - when Initial keys are discarded.

17.2.3. 0-RTT

A 0-RTT packet uses long headers with a type value of 0x1, followed by the Length and Packet Number fields. The first byte contains the Reserved and Packet Number Length bits. It is used to carry "early" data from the client to the server as part of the first flight, prior to handshake completion. As part of the TLS handshake, the server can accept or reject this early data.

See Section 2.3 of [TLS13] for a discussion of 0-RTT data and its limitations.

```

+-----+
| 1 | 1 | 1 | R | R | P | P |
+-----+
|                                     Version (32)                                     |
+-----+
| DCID Len (8) |
+-----+
| Destination Connection ID (0..160) ...
+-----+
| SCID Len (8) |
+-----+
| Source Connection ID (0..160) ...
+-----+
| Length (i) ...
+-----+
| Packet Number (8/16/24/32) ...
+-----+
| Payload (*) ...
+-----+

```

0-RTT Packet

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

After a client receives a Retry packet, 0-RTT packets are likely to have been lost or discarded by the server. A client **SHOULD** 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, since the keys used to protect 0-RTT packets will not change as a result of responding to a Retry packet. 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.

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 **MUST NOT** send 0-RTT packets once it starts processing 1-RTT packets from the server. This means that 0-RTT packets cannot contain any response to frames from 1-RTT packets. For instance, a

client cannot send an ACK frame in a 0-RTT packet, because that can only acknowledge a 1-RTT packet. An acknowledgment for a 1-RTT packet MUST be carried in a 1-RTT packet.

A server SHOULD treat a violation of remembered limits as a connection error of an appropriate type (for instance, a `FLOW_CONTROL_ERROR` for exceeding stream data limits).

17.2.4. Handshake Packet

A Handshake packet uses long headers with a type value of 0x2, followed by the Length and Packet Number fields. The first byte contains the Reserved and Packet Number Length bits. It is used to carry acknowledgments and cryptographic handshake messages from the server and client.

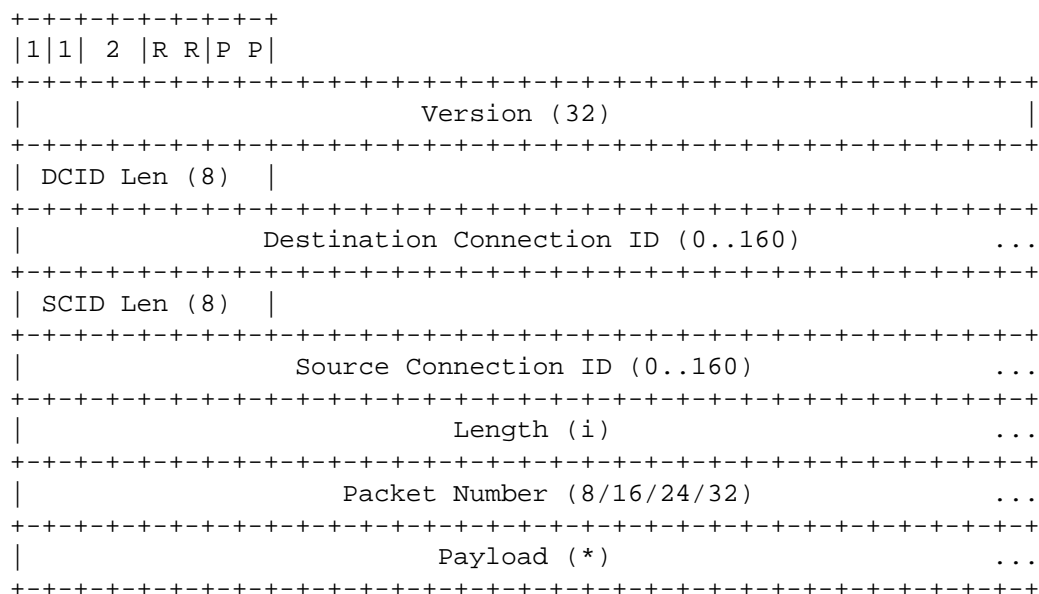


Figure 12: Handshake Protected Packet

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](#)).

Handshake packets are their own packet number space, and thus the first Handshake packet sent by a server contains a packet number of 0.

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

Like Initial packets (see [Section 17.2.2.1](#)), data in CRYPTO frames at the Handshake encryption level is discarded - and no longer retransmitted - when Handshake protection keys are discarded.

17.2.5. Retry Packet

A Retry packet uses a long packet header with a type value of 0x3. It carries an address validation token created by the server. It is used by a server that wishes to perform a retry (see [Section 8.1](#)).

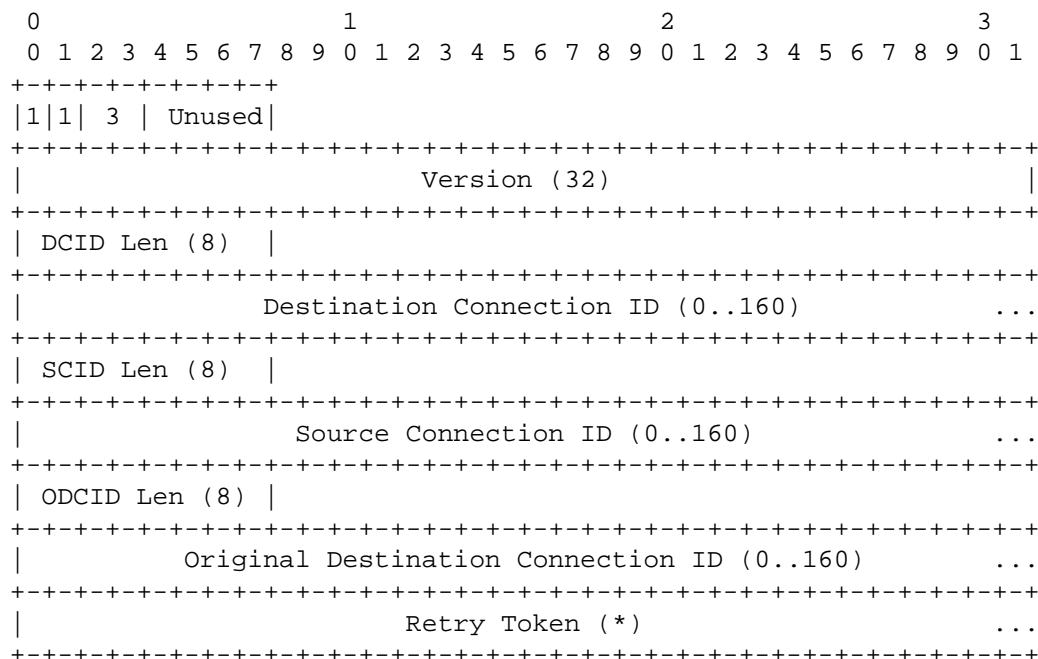


Figure 13: Retry Packet

A Retry packet (shown in Figure 13) does not contain any protected fields. The value in the Unused field is selected randomly by the server. In addition to the long header, it contains these additional fields:

ODCID Len: The ODCID Len contains the length in bytes of the Original Destination Connection ID field that follows it. This length is encoded as a 8-bit unsigned integer. In QUIC version 1, this value MUST NOT exceed 20 bytes. Clients that receive a version 1 Retry Packet with a value larger than 20 MUST drop the packet.

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 ODCID Len.

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 server MUST NOT send more than one Retry packet in response to a single UDP datagram.

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 8.1.3](#)).

The next Initial packet from the client uses the connection ID and token values from the Retry packet (see [Section 7.2](#)). Aside from this, the Initial packet sent by the client is subject to the same restrictions as the first Initial packet. A client MUST use the same cryptographic handshake message it includes in this packet. A server MAY treat a packet that contains a different cryptographic handshake message as a connection error or discard it.

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 MUST NOT change the cryptographic handshake message it sends in response to receiving a Retry.

A client MUST NOT reset the packet number for any packet number space after processing a Retry packet; [Section 17.2.3](#) contains more information on this.

A server acknowledges the use of a Retry packet for a connection using the `original_connection_id` transport parameter (see [Section 18.2](#)). 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 MUST validate that the `original_connection_id` transport parameter is present and correct; otherwise, it MUST validate 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.

[17.3](#). Short Header Packets

This version of QUIC defines a single packet type which uses the short packet header.

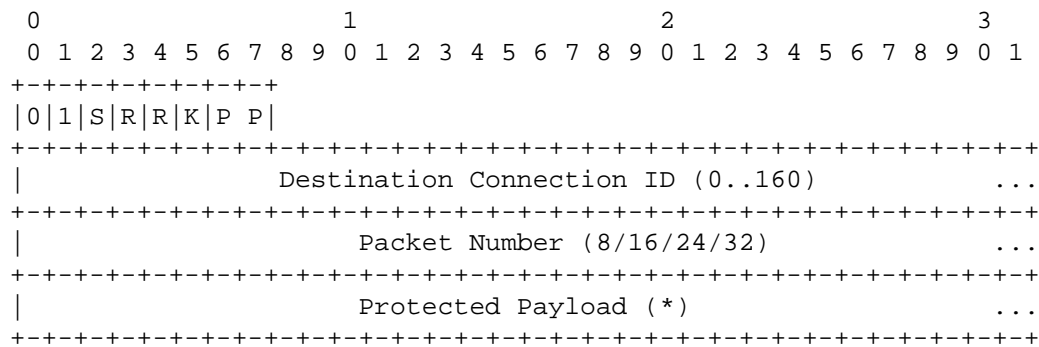


Figure 14: Short Header Packet Format

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 byte 0 is set to 0 for the short header.

Fixed Bit: The next bit (0x40) of byte 0 is set to 1. Packets containing a zero value for this bit are not valid packets in this version and **MUST** be discarded.

Spin Bit (S): The third most significant bit (0x20) of byte 0 is the latency spin bit, set as described in [Section 17.3.1](#).

Reserved Bits (R): The next two bits (those with a mask of 0x18) of byte 0 are reserved. These bits are protected using header protection (see Section 5.4 of [QUIC-TLS]). The value included prior to protection MUST be set to 0. An endpoint MUST treat receipt of a packet that has a non-zero value for these bits, after removing both packet and header protection, as a connection error of type `PROTOCOL_VIOLATION`. Discarding such a packet after only removing header protection can expose the endpoint to attacks (see Section 9.3 of [QUIC-TLS]).

Key Phase (K): The next bit (0x04) of byte 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. This bit is protected using header protection (see Section 5.4 of [QUIC-TLS]).

Packet Number Length (P): The least significant two bits (those with a mask of 0x03) of byte 0 contain the length of the packet number, encoded as an unsigned, two-bit integer that is one less than the length of the packet number field in bytes. That is, the length

of the packet number field is the value of this field, plus one. These bits are protected using header protection (see Section 5.4 of [QUIC-TLS]).

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 to 4 bytes long. The packet number has confidentiality protection separate from packet protection, as described in Section 5.4 of [QUIC-TLS]. The length of the packet number field is encoded in Packet Number Length field. See Section 17.1 for details.

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

The header form bit and the 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.

17.3.1. Latency Spin Bit

The latency spin bit enables passive latency monitoring from observation points on the network path throughout the duration of a connection. The spin bit is only present in the short packet header, since it is possible to measure the initial RTT of a connection by observing the handshake. Therefore, the spin bit is available after version negotiation and connection establishment are completed. On-path measurement and use of the latency spin bit is further discussed in [QUIC-MANAGEABILITY].

The spin bit is an OPTIONAL feature of QUIC. A QUIC stack that chooses to support the spin bit MUST implement it as specified in this section.

Each endpoint unilaterally decides if the spin bit is enabled or disabled for a connection. Implementations MUST allow administrators of clients and servers to disable the spin bit either globally or on a per-connection basis. Even when the spin bit is not disabled by the administrator, implementations MUST disable the spin bit for a given connection with a certain likelihood. The random selection process SHOULD be designed such that on average the spin bit is disabled for at least one eighth of network paths. The selection process performed at the beginning of the connection SHOULD be applied for all paths used by the connection.

When the spin bit is disabled, endpoints MAY set the spin bit to any value, and MUST ignore any incoming value. It is RECOMMENDED that endpoints set the spin bit to a random value either chosen independently for each packet or chosen independently for each connection ID.

If the spin bit is enabled for the connection, the endpoint maintains a spin value and sets the spin bit in the short header to the currently stored value when a packet with a short header is sent out. The spin value is initialized to 0 in the endpoint at connection start. Each endpoint also remembers the highest packet number seen from its peer on the connection.

When a server receives a short header packet that increments the highest packet number seen by the server from the client, it sets the spin value to be equal to the spin bit in the received packet.

When a client receives a short header packet that increments the highest packet number seen by the client from the server, it sets the spin value to the inverse of the spin bit in the received packet.

An endpoint resets its spin value to zero when sending the first packet of a given connection with a new connection ID. This reduces the risk that transient spin bit state can be used to link flows across connection migration or ID change.

With this mechanism, the server reflects the spin value received, while the client 'spins' it after one RTT. On-path observers can measure the time between two spin bit toggle events to estimate the end-to-end RTT of a connection.

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].

```

enum {
    original_connection_id(0),
    idle_timeout(1),
    stateless_reset_token(2),
    max_packet_size(3),
    initial_max_data(4),
    initial_max_stream_data_bidi_local(5),
    initial_max_stream_data_bidi_remote(6),
    initial_max_stream_data_uni(7),
    initial_max_streams_bidi(8),
    initial_max_streams_uni(9),
    ack_delay_exponent(10),
    max_ack_delay(11),
    disable_active_migration(12),
    preferred_address(13),
    active_connection_id_limit(14),
    (65535)
} TransportParameterId;

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

TransportParameter TransportParameters<0..2^16-1>;

```

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 bytes, which are then included in the cryptographic handshake.

18.1. Reserved Transport Parameters

Transport parameters with an identifier of the form "31 * N + 27" for integer values of N are reserved to exercise the requirement that unknown transport parameters be ignored. These transport parameters have no semantics, and may carry arbitrary values.

18.2. Transport Parameter Definitions

This section details the transport parameters defined in this document.

Many transport parameters listed here have integer values. Those transport parameters that are identified as integers use a variable-length integer encoding (see [Section 16](#)) and have a default value of 0 if the transport parameter is absent, unless otherwise stated.

The following transport parameters are defined:

`original_connection_id` (0x0000): The value of the Destination Connection ID field from the first Initial packet sent by the client. This transport parameter is only sent by a server. A server **MUST** include the `original_connection_id` transport parameter if it sent a Retry packet.

`idle_timeout` (0x0001): The idle timeout is a value in milliseconds that is encoded as an integer; see ([Section 10.2](#)). If this parameter is absent or zero then the idle timeout is disabled.

`stateless_reset_token` (0x0002): A stateless reset token is used in verifying a stateless reset; see [Section 10.4](#). This parameter is a sequence of 16 bytes. This transport parameter **MUST NOT** be sent by a client, but **MAY** be sent by a server. A server that does not send this transport parameter cannot use stateless reset ([Section 10.4](#)) for the connection ID negotiated during the handshake.

`max_packet_size` (0x0003): The maximum packet size parameter is an integer value that limits the size of packets that the endpoint is willing to receive. 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](#)).

`initial_max_data` (0x0004): The initial maximum data parameter is an integer value that contains the initial value for the maximum amount of data that can be sent on the connection. This is equivalent to sending a `MAX_DATA` ([Section 19.9](#)) for the connection immediately after completing the handshake.

`initial_max_stream_data_bidi_local` (0x0005): This parameter is an integer value specifying the initial flow control limit for locally-initiated bidirectional streams. This limit applies to newly created bidirectional streams opened by the endpoint that sends the transport parameter. In client transport parameters, this applies to streams with an identifier with the least significant two bits set to 0x0; in server transport parameters, this applies to streams with the least significant two bits set to 0x1.

`initial_max_stream_data_bidi_remote (0x0006)`: This parameter is an integer value specifying the initial flow control limit for peer-initiated bidirectional streams. This limit applies to newly created bidirectional streams opened by the endpoint that receives the transport parameter. In client transport parameters, this applies to streams with an identifier with the least significant two bits set to 0x1; in server transport parameters, this applies to streams with the least significant two bits set to 0x0.

`initial_max_stream_data_uni (0x0007)`: This parameter is an integer value specifying the initial flow control limit for unidirectional streams. This limit applies to newly created unidirectional streams opened by the endpoint that receives the transport parameter. In client transport parameters, this applies to streams with an identifier with the least significant two bits set to 0x3; in server transport parameters, this applies to streams with the least significant two bits set to 0x2.

`initial_max_streams_bidi (0x0008)`: The initial maximum bidirectional streams parameter is an integer value that contains the initial maximum number of bidirectional streams the peer may initiate. If this parameter is absent or zero, the peer cannot open bidirectional streams until a `MAX_STREAMS` frame is sent. Setting this parameter is equivalent to sending a `MAX_STREAMS` ([Section 19.11](#)) of the corresponding type with the same value.

`initial_max_streams_uni (0x0009)`: The initial maximum unidirectional streams parameter is an integer value that contains the initial maximum number of unidirectional streams the peer may initiate. If this parameter is absent or zero, the peer cannot open unidirectional streams until a `MAX_STREAMS` frame is sent. Setting this parameter is equivalent to sending a `MAX_STREAMS` ([Section 19.11](#)) of the corresponding type with the same value.

`ack_delay_exponent (0x000a)`: The ACK delay exponent is an integer value indicating an exponent used to decode the ACK Delay field in the ACK frame ([Section 19.3](#)). If this value is absent, a default value of 3 is assumed (indicating a multiplier of 8). Values above 20 are invalid.

`max_ack_delay (0x000b)`: The maximum ACK delay is an integer value indicating the maximum amount of time in milliseconds by which the endpoint will delay sending acknowledgments. This value SHOULD include the receiver's expected delays in alarms firing. For example, if a receiver sets a timer for 5ms and alarms commonly fire up to 1ms late, then it should send a `max_ack_delay` of 6ms. If this value is absent, a default of 25 milliseconds is assumed. Values of 2^{14} or greater are invalid.

`disable_active_migration (0x000c)`: The disable active migration transport parameter is included if the endpoint does not support active connection migration ([Section 9](#)). Peers of an endpoint that sets this transport parameter MUST NOT send any packets, including probing packets ([Section 9.1](#)), from a local address or port other than that used to perform the handshake. This parameter is a zero-length value.

`preferred_address (0x000d)`: 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](#). The format of this transport parameter is the PreferredAddress struct shown in Figure 16. This transport parameter is only sent by a server. Servers MAY choose to only send a preferred address of one address family by sending an all-zero address and port (0.0.0.0:0 or ::.0) for the other family. IP addresses are encoded in network byte order.

```
struct {  
    opaque ipv4Address[4];  
    uint16 ipv4Port;  
    opaque ipv6Address[16];  
    uint16 ipv6Port;  
    opaque connectionId<0..20>;  
    opaque statelessResetToken[16];  
} PreferredAddress;
```

Figure 16: Preferred Address format

`active_connection_id_limit (0x000e)`: The maximum number of connection IDs from the peer that an endpoint is willing to store. This value includes only connection IDs sent in NEW_CONNECTION_ID frames. If this parameter is absent, a default of 0 is assumed.

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 19.10](#)) 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.

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 byte that identifies the frame as a PADDING frame.

19.2. PING Frame

Endpoints can use PING frames (type=0x01) 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.

19.3. ACK Frames

Receivers send ACK frames (types 0x02 and 0x03) to inform senders of packets they have received and processed. The ACK frame contains one or more ACK Ranges. ACK Ranges identify acknowledged packets. If the frame type is 0x03, ACK frames also contain the sum of QUIC

packets with associated ECN marks received on the connection up until this point. QUIC implementations **MUST** properly handle both types and, if they have enabled ECN for packets they send, they **SHOULD** use the information in the ECN section to manage their congestion state.

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 as follows:

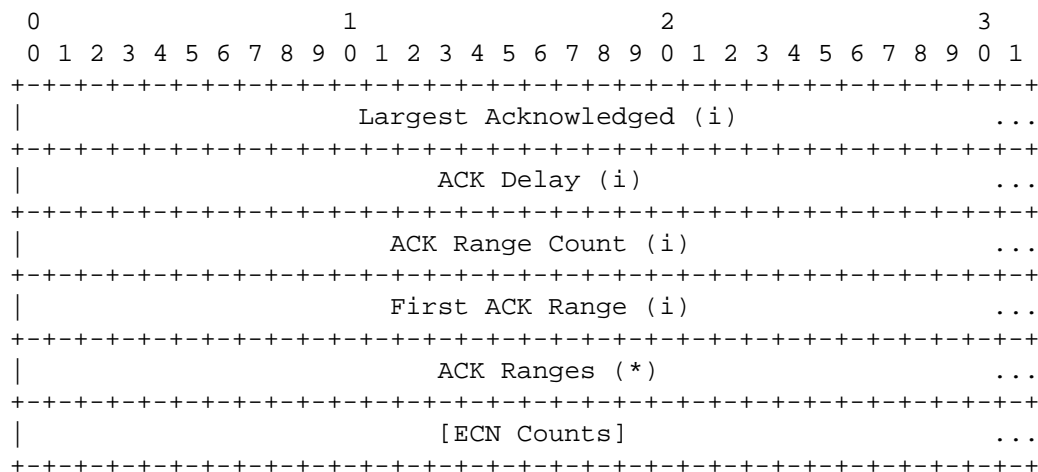


Figure 17: ACK Frame Format

ACK frames contain the following fields:

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 representing the time delta in microseconds between when this ACK was sent and when the largest acknowledged packet, as indicated in the Largest Acknowledged field, was received by this peer. 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 (see [Section 18.2](#)). Scaling in this fashion allows for a larger range of values with a shorter encoding at the cost of lower resolution. Because the receiver doesn't use the ACK Delay for Initial and Handshake packets, a sender SHOULD send a value of 0.

ACK Range Count: A variable-length integer specifying the number of Gap and ACK Range fields in the frame.

First ACK Range: A variable-length integer indicating the number of contiguous packets preceding the Largest Acknowledged that are being acknowledged. The First ACK Range is encoded as an ACK Range (see [Section 19.3.1](#)) starting from the Largest Acknowledged. That is, the smallest packet acknowledged in the range is determined by subtracting the First ACK Range value from the Largest Acknowledged.

ACK Ranges: Contains additional ranges of packets which are alternately not acknowledged (Gap) and acknowledged (ACK Range); see [Section 19.3.1](#).

ECN Counts: The three ECN Counts; see [Section 19.3.2](#).

[19.3.1](#). ACK Ranges

The ACK Ranges field consists of alternating Gap and ACK Range values in descending packet number order. The number of Gap and ACK Range values is determined by the ACK Range Count field; one of each value is present for each value in the ACK Range Count field.

ACK Ranges are structured as follows:

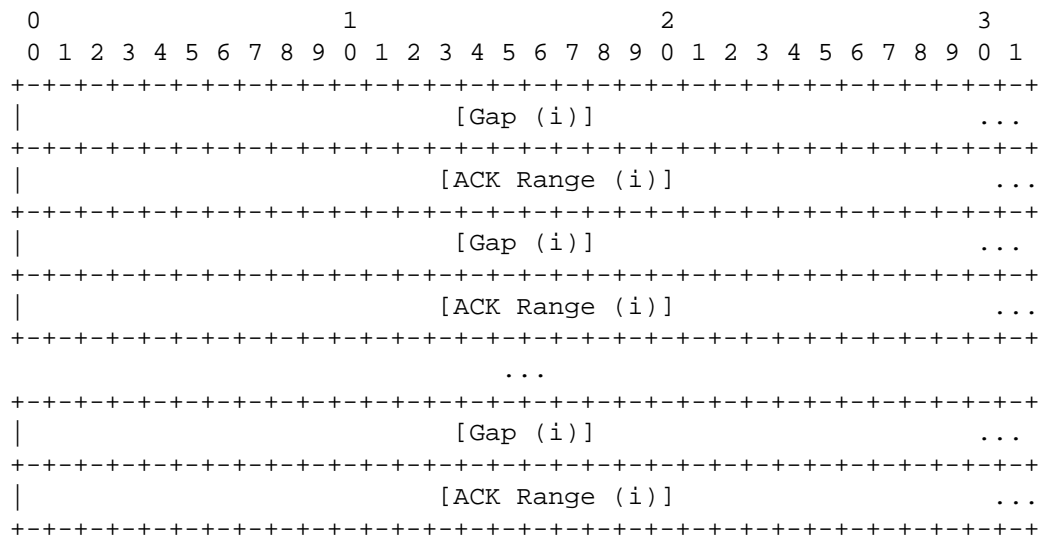


Figure 18: ACK Ranges

The fields that form the ACK Ranges are:

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 Range.

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

Gap and ACK Range value use a relative integer encoding for efficiency. Though each encoded value is positive, the values are subtracted, so that each ACK Range describes progressively lower-numbered packets.

Each ACK Range acknowledges a contiguous range of packets by indicating the number of acknowledged packets that precede the largest packet number in that range. A value of zero indicates that only the largest packet number is acknowledged. Larger ACK Range 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 range, the smallest value is determined by the formula:

$$\text{smallest} = \text{largest} - \text{ack_range}$$

An ACK Range acknowledges all packets between the smallest packet number and the largest, inclusive.

The largest value for an ACK Range is determined by cumulatively subtracting the size of all preceding ACK Ranges 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 subsequent ACK Range using the following formula:

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

If any computed packet number is negative, an endpoint MUST generate a connection error of type `FRAME_ENCODING_ERROR` indicating an error in an ACK frame.

19.3.2. ECN Counts

The ACK frame uses the least significant bit (that is, type 0x03) to indicate ECN feedback and report receipt of QUIC packets with associated ECN codepoints of ECT(0), ECT(1), or CE in the packet's IP header. ECN Counts are only present when the ACK frame type is 0x03.

ECN Counts are only parsed when the ACK frame type is 0x03. There are 3 ECN counts, as follows:

[illegible]

The three ECN Counts are:

ECT(0) Count: A variable-length integer representing the total number of packets received with the ECT(0) codepoint in the packet number space of the ACK frame.

ECT(1) Count: A variable-length integer representing the total number of packets received with the ECT(1) codepoint in the packet number space of the ACK frame.

CE Count: A variable-length integer representing the total number of packets received with the CE codepoint in the packet number space of the ACK frame.

ECN counts are maintained separately for each packet number space.

19.4. RESET_STREAM Frame

An endpoint uses a RESET_STREAM frame (type=0x04) to abruptly terminate the sending part of a stream.

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

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

The RESET_STREAM frame is as follows:

```

      0               1               2               3
      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+-----+-----+-----+-----+
|                               Stream ID (i)                               ...
+-----+-----+-----+-----+-----+-----+-----+-----+
|                               Application Error Code (i)                               ...
+-----+-----+-----+-----+-----+-----+-----+-----+
|                               Final Size (i)                               ...
+-----+-----+-----+-----+-----+-----+-----+-----+

```

RESET_STREAM frames contain the following fields:

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

Application Protocol Error Code: A variable-length integer containing the application protocol error code (see [Section 20.1](#)) which indicates why the stream is being closed.

Final Size: A variable-length integer indicating the final size of the stream by the RESET_STREAM sender, in unit of bytes.

19.5. STOP_SENDING Frame

An endpoint uses a STOP_SENDING frame (type=0x05) to communicate that incoming data is being discarded on receipt at application request. STOP_SENDING requests that a peer cease transmission on a stream.

A STOP_SENDING frame can be sent for streams in the Recv or Size Known states (see [Section 3.1](#)). Receiving a STOP_SENDING frame for a locally-initiated stream that has not yet been created MUST be treated as a connection error of type STREAM_STATE_ERROR. An endpoint that receives a STOP_SENDING frame for a receive-only stream MUST terminate the connection with error STREAM_STATE_ERROR.

The STOP_SENDING frame is as follows:

```

      0               1               2               3
      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+-----+-----+-----+-----+
|                               Stream ID (i)                               ...
+-----+-----+-----+-----+-----+-----+-----+-----+
|                               Application Error Code (i)                               ...
+-----+-----+-----+-----+-----+-----+-----+-----+

```

STOP_SENDING frames contain the following fields:

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

Application Error Code: A variable-length integer containing the application-specified reason the sender is ignoring the stream (see [Section 20.1](#)).

19.6. CRYPTO Frame

The CRYPTO frame (type=0x06) is used to transmit cryptographic handshake messages. It can be sent in all packet types except 0-RTT. 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.

The CRYPTO frame is as follows:

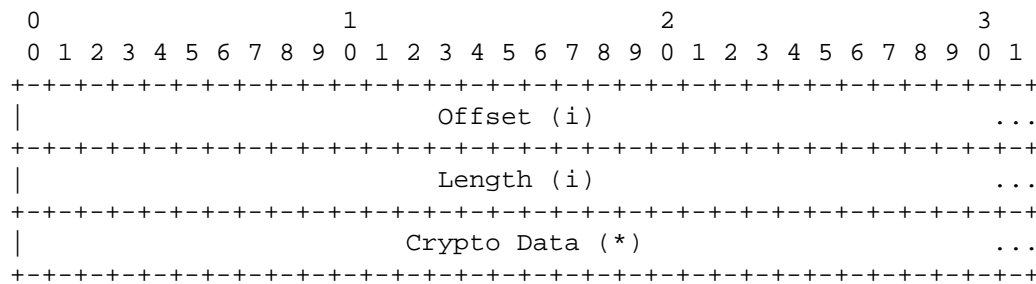


Figure 19: CRYPTO Frame Format

CRYPTO frames contain 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.

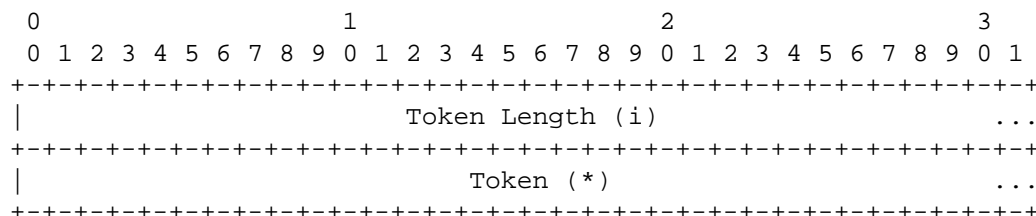
There is a separate flow of cryptographic handshake data in each encryption level, each of which starts at an offset of 0. This implies that each encryption level is treated as a separate CRYPTO stream of data.

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.

19.7. NEW_TOKEN Frame

A server sends a NEW_TOKEN frame (type=0x07) to provide the client with a token to send in the header of an Initial packet for a future connection.

The NEW_TOKEN frame is as follows:



NEW_TOKEN frames contain the following fields:

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.

An endpoint might receive multiple NEW_TOKEN frames that contain the same token value. Endpoints are responsible for discarding duplicate values, which might be used to link connection attempts; see [Section 8.1.2](#).

Clients MUST NOT send NEW_TOKEN frames. Servers MUST treat receipt of a NEW_TOKEN frame as a connection error of type `PROTOCOL_VIOLATION`.

19.8. STREAM Frames

STREAM frames implicitly create a stream and carry stream data. The STREAM frame takes the form `0b00001XXX` (or the set of values from `0x08` to `0x0f`). The value of the three low-order bits of the frame type determines the fields that are present in the frame.

- o 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 bytes of the stream, or the end of a stream that includes no data).
- o 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.
- o The FIN bit (`0x01`) of the frame type is set only on frames that contain the final size 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 `STREAM_STATE_ERROR`.

The STREAM frames are as follows:

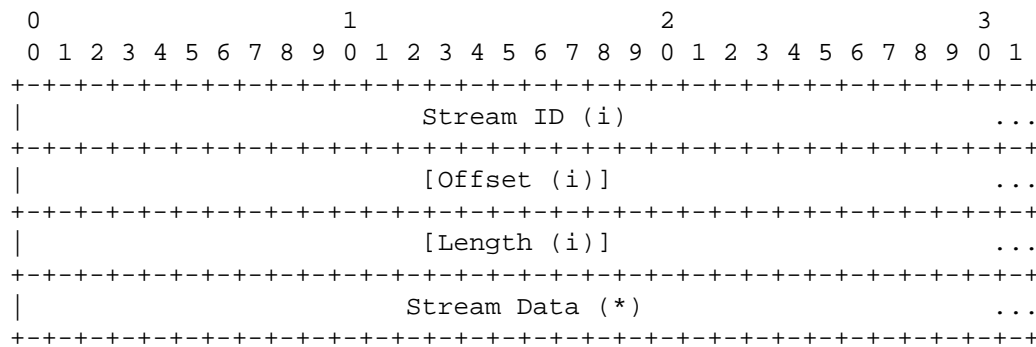


Figure 20: STREAM Frame Format

STREAM frames contain 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 bytes 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 offset and data length - cannot exceed $2^{62}-1$, as it is not possible to provide flow control credit for that data. Receipt of a frame that exceeds this limit will be treated as a connection error of type `FLOW_CONTROL_ERROR`.

19.9. MAX_DATA Frame

The MAX_DATA frame (type=0x10) 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 MAX_DATA frame is as follows:

```

      0               1               2               3
      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+-----+-----+-----+-----+
|                                     Maximum Data (i)                               ...
+-----+-----+-----+-----+-----+-----+-----+-----+

```

MAX_DATA frames contain the following fields:

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

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](#)).

19.10. MAX_STREAM_DATA Frame

The MAX_STREAM_DATA frame (type=0x11) is used in flow control to inform a peer of the maximum amount of data that can be sent on a stream.

A MAX_STREAM_DATA frame can be sent for streams in the Recv state (see [Section 3.1](#)). Receiving a MAX_STREAM_DATA frame for a locally-initiated stream that has not yet been created MUST be treated as a connection error of type `STREAM_STATE_ERROR`. An endpoint that receives a MAX_STREAM_DATA frame for a receive-only stream MUST terminate the connection with error `STREAM_STATE_ERROR`.

The MAX_STREAM_DATA frame is as follows:

```

      0               1               2               3
      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+-----+-----+-----+-----+
|                                     Stream ID (i)                               ...
+-----+-----+-----+-----+-----+-----+-----+-----+
|                                     Maximum Stream Data (i)                       ...
+-----+-----+-----+-----+-----+-----+-----+-----+

```

MAX_STREAM_DATA frames contain the following fields:

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 bytes.

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](#)).

19.11. MAX_STREAMS Frames

The MAX_STREAMS frames (type=0x12 and 0x13) inform the peer of the cumulative number of streams of a given type it is permitted to open. A MAX_STREAMS frame with a type of 0x12 applies to bidirectional streams, and a MAX_STREAMS frame with a type of 0x13 applies to unidirectional streams.

The MAX_STREAMS frames are as follows:

```

      0               1               2               3
      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                                     Maximum Streams (i)                               ...
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

```

MAX_STREAMS frames contain the following fields:

Maximum Streams: A count of the cumulative number of streams of the corresponding type that can be opened over the lifetime of the connection.

Loss or reordering can cause a MAX_STREAMS frame to be received which states a lower stream limit than an endpoint has previously received. MAX_STREAMS frames which do not increase the stream limit **MUST** be ignored.

An endpoint **MUST NOT** open more streams than permitted by the current stream limit set by its peer. For instance, a server that receives a unidirectional stream limit of 3 is permitted to open stream 3, 7,

and 11, but not stream 15. An endpoint MUST terminate a connection with a `STREAM_LIMIT_ERROR` error if a peer opens more streams than was permitted.

Note that these frames (and the corresponding transport parameters) do not describe the number of streams that can be opened concurrently. The limit includes streams that have been closed as well as those that are open.

19.12. DATA_BLOCKED Frame

A sender SHOULD send a `DATA_BLOCKED` frame (type=0x14) when it wishes to send data, but is unable to due to connection-level flow control (see [Section 4](#)). `DATA_BLOCKED` frames can be used as input to tuning of flow control algorithms (see [Section 4.2](#)).

The `DATA_BLOCKED` frame is as follows:

```

      0               1               2               3
      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+-----+-----+-----+-----+
|                                     Data Limit (i)                                     ...
+-----+-----+-----+-----+-----+-----+-----+-----+

```

`DATA_BLOCKED` frames contain the following fields:

Data Limit: A variable-length integer indicating the connection-level limit at which blocking occurred.

19.13. STREAM_DATA_BLOCKED Frame

A sender SHOULD send a `STREAM_DATA_BLOCKED` frame (type=0x15) when it wishes to send data, but is unable to due to stream-level flow control. This frame is analogous to `DATA_BLOCKED` ([Section 19.12](#)).

An endpoint that receives a `STREAM_DATA_BLOCKED` frame for a send-only stream MUST terminate the connection with error `STREAM_STATE_ERROR`.

The `STREAM_DATA_BLOCKED` frame is as follows:

```

      0               1               2               3
      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+-----+-----+-----+-----+
|                                     Stream ID (i)                                     ...
+-----+-----+-----+-----+-----+-----+-----+-----+
|                                     Stream Data Limit (i)                               ...
+-----+-----+-----+-----+-----+-----+-----+-----+

```

STREAM_DATA_BLOCKED frames contain the following fields:

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

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

19.14. STREAMS_BLOCKED Frames

A sender SHOULD send a STREAMS_BLOCKED frame (type=0x16 or 0x17) when it wishes to open a stream, but is unable to due to the maximum stream limit set by its peer (see [Section 19.11](#)). A STREAMS_BLOCKED frame of type 0x16 is used to indicate reaching the bidirectional stream limit, and a STREAMS_BLOCKED frame of type 0x17 indicates reaching the unidirectional stream limit.

A STREAMS_BLOCKED frame does not open the stream, but informs the peer that a new stream was needed and the stream limit prevented the creation of the stream.

The STREAMS_BLOCKED frames are as follows:

```

0                               1                               2                               3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                               Stream Limit (i)                               ...
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

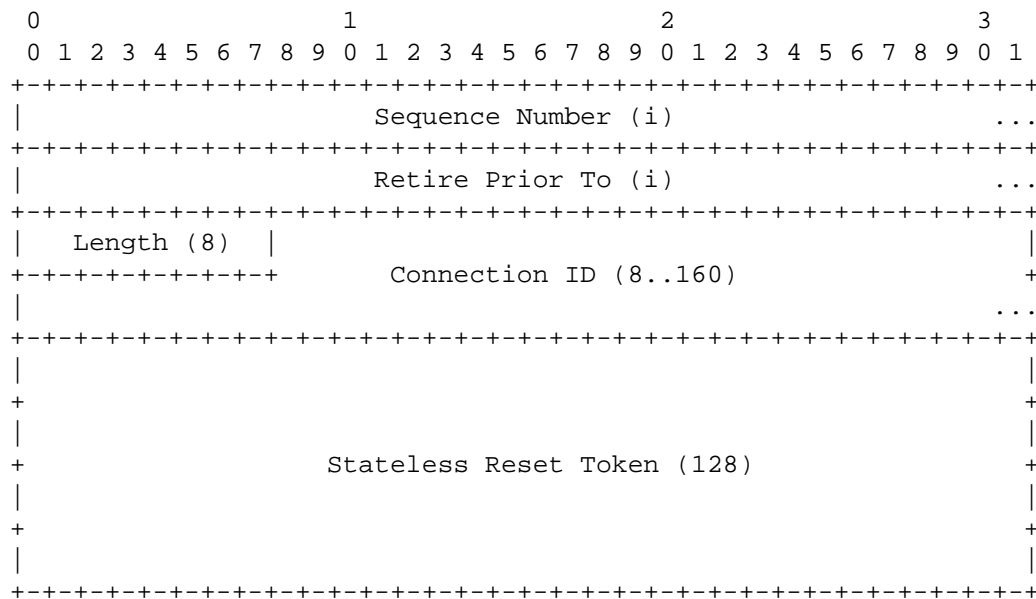
STREAMS_BLOCKED frames contain the following fields:

Stream Limit: A variable-length integer indicating the stream limit at the time the frame was sent.

19.15. NEW_CONNECTION_ID Frame

An endpoint sends a NEW_CONNECTION_ID frame (type=0x18) 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:



NEW_CONNECTION_ID frames contain the following fields:

Sequence Number: The sequence number assigned to the connection ID by the sender. See [Section 5.1.1](#).

Retire Prior To: A variable-length integer indicating which connection IDs should be retired. See [Section 5.1.2](#).

Length: An 8-bit unsigned integer containing the length of the connection ID. Values less than 1 and greater than 20 are invalid and MUST be treated as a connection error of type `PROTOCOL_VIOLATION`.

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, or if a sequence number is used for different connection IDs, the endpoint **MAY** treat that receipt as a connection error of type `PROTOCOL_VIOLATION`.

The `Retire Prior To` field is a request for the peer to retire all connection IDs with a sequence number less than the specified value. This includes the initial and preferred_address transport parameter connection IDs. The peer **SHOULD** retire the corresponding connection IDs and send the corresponding `RETIRE_CONNECTION_ID` frames in a timely manner.

The `Retire Prior To` field **MUST** be less than or equal to the `Sequence Number` field. Receiving a value greater than the `Sequence Number` **MUST** be treated as a connection error of type `PROTOCOL_VIOLATION`.

Once a sender indicates a `Retire Prior To` value, smaller values sent in subsequent `NEW_CONNECTION_ID` frames have no effect. A receiver **MUST** ignore any `Retire Prior To` fields that do not increase the largest received `Retire Prior To` value.

19.16. `RETIRE_CONNECTION_ID` Frame

An endpoint sends a `RETIRE_CONNECTION_ID` frame (type=0x19) 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 19.15](#)).

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

The `RETIRE_CONNECTION_ID` frame is as follows:

```

      0                               1                               2                               3
      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|                               Sequence Number (i)                               ...
+-----+-----+-----+-----+-----+-----+-----+-----+-----+

```

RETIRE_CONNECTION_ID frames contain the following fields:

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`.

The sequence number specified in a RETIRE_CONNECTION_ID frame MUST NOT refer to the Destination Connection ID field of the packet in which the frame is contained. The peer MAY treat this 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`.

19.17. PATH_CHALLENGE Frame

Endpoints can use PATH_CHALLENGE frames (`type=0x1a`) to check reachability to the peer and for path validation during connection migration.

The PATH_CHALLENGE frames are as follows:

```

      0               1               2               3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                                     |
+                                     +
|                                     |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

```

Data (64)

PATH_CHALLENGE frames contain the following fields:

Data: This 8-byte field contains arbitrary data.

A PATH_CHALLENGE frame containing 8 bytes 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 19.18](#)) containing the same Data.

19.18. PATH_RESPONSE Frame

The PATH_RESPONSE frame (type=0x1b) is sent in response to a PATH_CHALLENGE frame. Its format is identical to the PATH_CHALLENGE frame ([Section 19.17](#)).

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`.

19.19. CONNECTION_CLOSE Frames

An endpoint sends a CONNECTION_CLOSE frame (type=0x1c or 0x1d) to notify its peer that the connection is being closed. The CONNECTION_CLOSE with a frame type of 0x1c is used to signal errors at only the QUIC layer, or the absence of errors (with the `NO_ERROR` code). The CONNECTION_CLOSE frame with a type of 0x1d is used to signal an error with the application that uses QUIC.

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

The CONNECTION_CLOSE frames are as follows:

```

0                               1                               2                               3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                               Error Code (i)                               ...
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                               [ Frame Type (i) ]                               ...
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                               Reason Phrase Length (i)                               ...
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                               Reason Phrase (*)                               ...
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

```

CONNECTION_CLOSE frames contain the following fields:

Error Code: A variable length integer error code which indicates the reason for closing this connection. A CONNECTION_CLOSE frame of type 0x1c uses codes from the space defined in [Section 20](#). A CONNECTION_CLOSE frame of type 0x1d uses codes from the application protocol error code space; see [Section 20.1](#)

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. The

application-specific variant of CONNECTION_CLOSE (type 0x1d) does not include this field.

Reason Phrase Length: A variable-length integer specifying the length of the reason phrase in bytes. Because a CONNECTION_CLOSE frame cannot be split between packets, 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].

19.20. 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 62-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_LIMIT_ERROR (0x4): An endpoint received a frame for a stream identifier that exceeded its advertised stream limit for the corresponding stream type.

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_SIZE_ERROR (0x6): An endpoint received a STREAM frame containing data that exceeded the previously established final size. Or an endpoint received a STREAM frame or a RESET_STREAM frame containing a final size that was lower than the size of stream data that was already received. Or an endpoint received a STREAM frame or a RESET_STREAM frame containing a different final size 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.

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

CRYPTO_BUFFER_EXCEEDED (0xD): An endpoint has received more data in CRYPTO frames than it can buffer.

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.

In defining these error codes, several principles are applied. Error conditions that might require specific action on the part of a recipient are given unique codes. Errors that represent common conditions are given specific codes. Absent either of these conditions, error codes are used to identify a general function of the stack, like flow control or transport parameter handling. Finally, generic errors are provided for conditions where implementations are unable or unwilling to use more specific codes.

20.1. Application Protocol Error Codes

Application protocol error codes are 62-bit unsigned integers, but the management of application error codes is left to application protocols. Application protocol error codes are used for the RESET_STREAM frame ([Section 19.4](#)), the STOP_SENDING frame ([Section 19.5](#)), and the CONNECTION_CLOSE frame with a type of 0x1d ([Section 19.19](#)).

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 14.2](#)), 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.

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. Amplification Attack

An attacker might be able to receive an address validation token ([Section 8](#)) from a server and then release the IP address it used to acquire that token. At a later time, the attacker may initiate a 0-RTT connection with a server by spoofing this same address, which might now address a different (victim) endpoint. The attacker can thus potentially cause the server to send an initial congestion window's worth of data towards the victim.

Servers SHOULD provide mitigations for this attack by limiting the usage and lifetime of address validation tokens (see [Section 8.1.2](#)).

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, loss or reordering may cause STREAM frames that open streams to be received out of sequence. On receiving a higher-numbered stream ID, a receiver is required to open all intervening streams of the same type (see [Section 3.2](#)). Thus, on a new connection, opening stream 4000000 opens 1 million and 1 client-initiated bidirectional streams.

The number of active streams is limited by the `initial_max_streams_bidi` and `initial_max_streams_uni` transport parameters, as explained in [Section 4.5](#). If chosen judiciously, these limits mitigate 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. Peer Denial of Service

QUIC and TLS both contain messages that have legitimate uses in some contexts, but that can be abused to cause a peer to expend processing resources without having any observable impact on the state of the connection.

Messages can also be used to change and revert state in small or inconsequential ways, such as by sending small increments to flow control limits.

If processing costs are disproportionately large in comparison to bandwidth consumption or effect on state, then this could allow a malicious peer to exhaust processing capacity.

While there are legitimate uses for all messages, implementations SHOULD track cost of processing relative to progress and treat excessive quantities of any non-productive packets as indicative of an attack. Endpoints MAY respond to this condition with a connection error, or by dropping packets.

21.8. 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 endpoints ignore the ECN codepoint field on

an IP packet unless at least one QUIC packet in that IP packet is successfully processed; see [Section 13.4](#).

21.9. 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.

21.10. Version Downgrade

This document defines QUIC Version Negotiation packets [Section 6](#), which can be used to negotiate the QUIC version used between two endpoints. However, this document does not specify how this negotiation will be performed between this version and subsequent future versions. In particular, Version Negotiation packets do not contain any mechanism to prevent version downgrade attacks. Future versions of QUIC that use Version Negotiation packets MUST define a mechanism that is robust against version downgrade attacks.

21.11. Targeted Attacks by Routing

Deployments should limit the ability of an attacker to target a new connection to a particular server instance. This means that client-controlled fields, such as the initial Destination Connection ID used on Initial and 0-RTT packets SHOULD NOT be used by themselves to make routing decisions. Ideally, routing decisions are made independently

of client-selected values; a Source Connection ID can be selected to route later packets to the same server.

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 6.

| Value | Parameter Name | Specification |
|--------|-------------------------------------|------------------------------|
| 0x0000 | original_connection_id | Section 18.2 |
| 0x0001 | idle_timeout | Section 18.2 |
| 0x0002 | stateless_reset_token | Section 18.2 |
| 0x0003 | max_packet_size | Section 18.2 |
| 0x0004 | initial_max_data | Section 18.2 |
| 0x0005 | initial_max_stream_data_bidi_local | Section 18.2 |
| 0x0006 | initial_max_stream_data_bidi_remote | Section 18.2 |
| 0x0007 | initial_max_stream_data_uni | Section 18.2 |
| 0x0008 | initial_max_streams_bidi | Section 18.2 |
| 0x0009 | initial_max_streams_uni | Section 18.2 |
| 0x000a | ack_delay_exponent | Section 18.2 |
| 0x000b | max_ack_delay | Section 18.2 |
| 0x000c | disable_active_migration | Section 18.2 |
| 0x000d | preferred_address | Section 18.2 |
| 0x000e | active_connection_id_limit | Section 18.2 |

Table 6: Initial QUIC Transport Parameters Entries

Additionally, each value of the format "31 * N + 27" for integer values of N (that is, "27", "58", "89", ...) MUST NOT be assigned by IANA.

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 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 0x0000 and 0x3fff).

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 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 | 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_LIMIT_ERROR | Too many streams opened | Section 20 |
| 0x5 | STREAM_STATE_ERROR | Frame received in invalid stream state | Section 20 |
| 0x6 | FINAL_SIZE_ERROR | Change to final size | Section 20 |
| 0x7 | FRAME_ENCODING_ERROR | Frame encoding error | Section 20 |
| 0x8 | TRANSPORT_PARAMETER_ERROR | Error in transport parameters | Section 20 |
| 0xA | PROTOCOL_VIOLATION | Generic protocol violation | Section 20 |
| 0xD | CRYPTO_BUFFER_EXCEEDED | CRYPTO data buffer overflowed | Section 20 |

Table 7: Initial QUIC Transport Error Codes Entries

23. References

23.1. Normative References

[DPLPMTUD]

Fairhurst, G., Jones, T., Tuexen, M., Ruengeler, I., and T. Voelker, "Packetization Layer Path MTU Discovery for Datagram Transports", [draft-ietf-tsvwg-datagram-plpmtud-08](#) (work in progress), June 2019.

[QUIC-RECOVERY]

Iyengar, J., Ed. and I. Swett, Ed., "QUIC Loss Detection and Congestion Control", [draft-ietf-quic-recovery-23](#) (work in progress), September 2019.

[QUIC-TLS]

Thomson, M., Ed. and S. Turner, Ed., "Using Transport Layer Security (TLS) to Secure QUIC", [draft-ietf-quic-tls-23](#) (work in progress), September 2019.

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

[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>.

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", [RFC 5116](#), DOI 10.17487/RFC5116, January 2008, <https://www.rfc-editor.org/info/rfc5116>.

- [RFC6437] Amante, S., Carpenter, B., Jiang, S., and J. Rajahalme, "IPv6 Flow Label Specification", [RFC 6437](#), DOI 10.17487/RFC6437, November 2011, <<https://www.rfc-editor.org/info/rfc6437>>.
- [RFC8085] Eggert, L., Fairhurst, G., and G. Shepherd, "UDP Usage Guidelines", [BCP 145](#), [RFC 8085](#), DOI 10.17487/RFC8085, March 2017, <<https://www.rfc-editor.org/info/rfc8085>>.
- [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>>.
- [RFC8201] 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>>.
- [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-07](#) (work in progress), September 2019.

[QUIC-MANAGEABILITY]

Kuehlewind, M. and B. Trammell, "Manageability of the QUIC Transport Protocol", [draft-ietf-quic-manageability-05](#) (work in progress), July 2019.

- [RFC1812] Baker, F., Ed., "Requirements for IP Version 4 Routers", [RFC 1812](#), DOI 10.17487/RFC1812, June 1995, <https://www.rfc-editor.org/info/rfc1812>.
- [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>.
- [RFC4303] Kent, S., "IP Encapsulating Security Payload (ESP)", [RFC 4303](#), DOI 10.17487/RFC4303, December 2005, <https://www.rfc-editor.org/info/rfc4303>.
- [RFC4443] Conta, A., Deering, S., and M. Gupta, Ed., "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification", STD 89, [RFC 4443](#), DOI 10.17487/RFC4443, March 2006, <https://www.rfc-editor.org/info/rfc4443>.
- [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>.
- [RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", [RFC 5681](#), DOI 10.17487/RFC5681, September 2009, <https://www.rfc-editor.org/info/rfc5681>.
- [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>>.
- [RFC8200] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", STD 86, [RFC 8200](#), DOI 10.17487/RFC8200, July 2017, <<https://www.rfc-editor.org/info/rfc8200>>.
- [SLOWLORIS]
RSnake Hansen, R., "Welcome to Slowloris...", June 2009, <<https://web.archive.org/web/20150315054838/http://ha.ckers.org/slowloris/>>.

Appendix A. Sample Packet Number Decoding Algorithm

The following pseudo-code shows how an implementation can decode packet numbers after header 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-22](#)

- o Rules for preventing correlation by connection ID tightened (#2084, #2929)
- o Clarified use of CONNECTION_CLOSE in Handshake packets (#2151, #2541, #2688)
- o Discourage regressions of largest acknowledged in ACK (#2205, #2752)
- o Improved robustness of validation process for ECN counts (#2534, #2752)
- o Require endpoints to ignore spurious migration attempts (#2342, #2893)
- o Transport parameter for disabling migration clarified to allow NAT rebinding (#2389, #2893)
- o Document principles for defining new error codes (#2388, #2880)
- o Reserve transport parameters for greasing (#2550, #2873)
- o A maximum ACK delay of 0 is used for handshake packet number spaces (#2646, #2638)
- o Improved rules for use of congestion control state on new paths (#2685, #2918)
- o Removed recommendation to coordinate spin for multiple connections that share a path (#2763, #2882)
- o Allow smaller stateless resets and recommend a smaller minimum on packets that might trigger a stateless reset (#2770, #2869, #2927)
- o Provide guidance around the interface to QUIC as used by application protocols (#2805, #2857)
- o Frames other than STREAM can cause STREAM_LIMIT_ERROR (#2825, #2826)

- o Tighter rules about processing of rejected 0-RTT packets (#2829, #2840, #2841)
- o Explanation of the effect of Retry on 0-RTT packets (#2842, #2852)
- o Cryptographic handshake needs to provide server transport parameter encryption (#2920, #2921)
- o Moved ACK generation guidance from recovery draft to transport draft (#1860, #2916).

B.2. Since [draft-ietf-quic-transport-21](#)

- o Connection ID lengths are now one octet, but limited in version 1 to 20 octets of length (#2736, #2749)

B.3. Since [draft-ietf-quic-transport-20](#)

- o Error codes are encoded as variable-length integers (#2672, #2680)
- o NEW_CONNECTION_ID includes a request to retire old connection IDs (#2645, #2769)
- o Tighter rules for generating and explicitly eliciting ACK frames (#2546, #2794)
- o Recommend having only one packet per encryption level in a datagram (#2308, #2747)
- o More normative language about use of stateless reset (#2471, #2574)
- o Allow reuse of stateless reset tokens (#2732, #2733)
- o Allow, but not require, enforcing non-duplicate transport parameters (#2689, #2691)
- o Added an active_connection_id_limit transport parameter (#1994, #1998)
- o max_ack_delay transport parameter defaults to 0 (#2638, #2646)
- o When sending 0-RTT, only remembered transport parameters apply (#2458, #2360, #2466, #2461)
- o Define handshake completion and confirmation; define clearer rules when it encryption keys should be discarded (#2214, #2267, #2673)

- o Prohibit path migration prior to handshake confirmation (#2309, #2370)
- o PATH_RESPONSE no longer needs to be received on the validated path (#2582, #2580, #2579, #2637)
- o PATH_RESPONSE frames are not stored and retransmitted (#2724, #2729)
- o Document hack for enabling routing of ICMP when doing PMTU probing (#1243, #2402)

B.4. Since [draft-ietf-quic-transport-19](#)

- o Refine discussion of 0-RTT transport parameters (#2467, #2464)
- o Fewer transport parameters need to be remembered for 0-RTT (#2624, #2467)
- o Spin bit text incorporated (#2564)
- o Close the connection when maximum stream ID in MAX_STREAMS exceeds $2^{62} - 1$ (#2499, #2487)
- o New connection ID required for intentional migration (#2414, #2413)
- o Connection ID issuance can be rate-limited (#2436, #2428)
- o The "QUIC bit" is ignored in Version Negotiation (#2400, #2561)
- o Initial packets from clients need to be padded to 1200 unless a Handshake packet is sent as well (#2522, #2523)
- o CRYPTO frames can be discarded if too much data is buffered (#1834, #2524)
- o Stateless reset uses a short header packet (#2599, #2600)

B.5. Since [draft-ietf-quic-transport-18](#)

- o Removed version negotiation; version negotiation, including authentication of the result, will be addressed in the next version of QUIC (#1773, #2313)
- o Added discussion of the use of IPv6 flow labels (#2348, #2399)

- o A connection ID can't be retired in a packet that uses that connection ID (#2101, #2420)
- o Idle timeout transport parameter is in milliseconds (from seconds) (#2453, #2454)
- o Endpoints are required to use new connection IDs when they use new network paths (#2413, #2414)
- o Increased the set of permissible frames in 0-RTT (#2344, #2355)

B.6. Since [draft-ietf-quic-transport-17](#)

- o Stream-related errors now use `STREAM_STATE_ERROR` (#2305)
- o Endpoints discard initial keys as soon as handshake keys are available (#1951, #2045)
- o Expanded conditions for ignoring ICMP packet too big messages (#2108, #2161)
- o Remove rate control from `PATH_CHALLENGE/PATH_RESPONSE` (#2129, #2241)
- o Endpoints are permitted to discard malformed initial packets (#2141)
- o Clarified ECN implementation and usage requirements (#2156, #2201)
- o Disable ECN count verification for packets that arrive out of order (#2198, #2215)
- o Use Probe Timeout (PTO) instead of RTO (#2206, #2238)
- o Loosen constraints on retransmission of ACK ranges (#2199, #2245)
- o Limit Retry and Version Negotiation to once per datagram (#2259, #2303)
- o Set a maximum value for `max_ack_delay` transport parameter (#2282, #2301)
- o Allow server preferred address for both IPv4 and IPv6 (#2122, #2296)
- o Corrected requirements for migration to a preferred address (#2146, #2349)

- o ACK of non-existent packet is illegal (#2298, #2302)

B.7. Since [draft-ietf-quic-transport-16](#)

- o Stream limits are defined as counts, not maximums (#1850, #1906)
- o Require amplification attack defense after closing (#1905, #1911)
- o Remove reservation of application error code 0 for STOPPING (#1804, #1922)
- o Renumbered frames (#1945)
- o Renumbered transport parameters (#1946)
- o Numeric transport parameters are expressed as varints (#1608, #1947, #1955)
- o Reorder the NEW_CONNECTION_ID frame (#1952, #1963)
- o Rework the first byte (#2006)
 - * Fix the 0x40 bit
 - * Change type values for long header
 - * Add spin bit to short header (#631, #1988)
 - * Encrypt the remainder of the first byte (#1322)
 - * Move packet number length to first byte
 - * Move ODCIL to first byte of retry packets
 - * Simplify packet number protection (#1575)
- o Allow STOP_SENDING to open a remote bidirectional stream (#1797, #2013)
- o Added mitigation for off-path migration attacks (#1278, #1749, #2033)
- o Don't let the PMTU to drop below 1280 (#2063, #2069)
- o Require peers to replace retired connection IDs (#2085)
- o Servers are required to ignore Version Negotiation packets (#2088)

- o Tokens are repeated in all Initial packets (#2089)
- o Clarified how PING frames are sent after loss (#2094)
- o Initial keys are discarded once Handshake are available (#1951, #2045)
- o ICMP PTB validation clarifications (#2161, #2109, #2108)

B.8. Since [draft-ietf-quic-transport-15](#)

Substantial editorial reorganization; no technical changes.

B.9. Since [draft-ietf-quic-transport-14](#)

- o Merge ACK and ACK_ECN (#1778, #1801)
- o Explicitly communicate max_ack_delay (#981, #1781)
- o Validate original connection ID after Retry packets (#1710, #1486, #1793)
- o Idle timeout is optional and has no specified maximum (#1765)
- o Update connection ID handling; add RETIRE_CONNECTION_ID type (#1464, #1468, #1483, #1484, #1486, #1495, #1729, #1742, #1799, #1821)
- o Include a Token in all Initial packets (#1649, #1794)
- o Prevent handshake deadlock (#1764, #1824)

B.10. Since [draft-ietf-quic-transport-13](#)

- o Streams open when higher-numbered streams of the same type open (#1342, #1549)
- o Split initial stream flow control limit into 3 transport parameters (#1016, #1542)
- o All flow control transport parameters are optional (#1610)
- o Removed UNSOLICITED_PATH_RESPONSE error code (#1265, #1539)
- o Permit stateless reset in response to any packet (#1348, #1553)
- o Recommended defense against stateless reset spoofing (#1386, #1554)

- o Prevent infinite stateless reset exchanges (#1443, #1627)
- o Forbid processing of the same packet number twice (#1405, #1624)
- o Added a packet number decoding example (#1493)
- o More precisely define idle timeout (#1429, #1614, #1652)
- o Corrected format of Retry packet and prevented looping (#1492, #1451, #1448, #1498)
- o Permit 0-RTT after receiving Version Negotiation or Retry (#1507, #1514, #1621)
- o Permit Retry in response to 0-RTT (#1547, #1552)
- o Looser verification of ECN counters to account for ACK loss (#1555, #1481, #1565)
- o Remove frame type field from APPLICATION_CLOSE (#1508, #1528)

B.11. Since [draft-ietf-quic-transport-12](#)

- o 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)
- o Enable server to transition connections to a preferred address (#560, #1251, #1373)
- o Added ECN feedback mechanisms and handling; new ACK_ECN frame (#804, #805, #1372)
- o Changed rules and recommendations for use of new connection IDs (#1258, #1264, #1276, #1280, #1419, #1452, #1453, #1465)

- o Added a transport parameter to disable intentional connection migration (#1271, #1447)
- o Packets from different connection ID can't be coalesced (#1287, #1423)
- o 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)
- o Stateless Reset is now symmetric and subject to size constraints (#466, #1346)
- o Added frame type extension mechanism (#58, #1473)

B.12. Since [draft-ietf-quic-transport-11](#)

- o Enable server to transition connections to a preferred address (#560, #1251)
- o Packet numbers are encrypted (#1174, #1043, #1048, #1034, #850, #990, #734, #1317, #1267, #1079)
- o Packet numbers use a variable-length encoding (#989, #1334)
- o STREAM frames can now be empty (#1350)

B.13. Since [draft-ietf-quic-transport-10](#)

- o Swap payload length and packed number fields in long header (#1294)
- o Clarified that CONNECTION_CLOSE is allowed in Handshake packet (#1274)
- o Spin bit reserved (#1283)
- o Coalescing multiple QUIC packets in a UDP datagram (#1262, #1285)
- o A more complete connection migration (#1249)
- o Refine opportunistic ACK defense text (#305, #1030, #1185)
- o A Stateless Reset Token isn't mandatory (#818, #1191)
- o Removed implicit stream opening (#896, #1193)

- o An empty STREAM frame can be used to open a stream without sending data (#901, #1194)
- o Define stream counts in transport parameters rather than a maximum stream ID (#1023, #1065)
- o STOP_SENDING is now prohibited before streams are used (#1050)
- o Recommend including ACK in Retry packets and allow PADDING (#1067, #882)
- o Endpoints now become closing after an idle timeout (#1178, #1179)
- o Remove implication that Version Negotiation is sent when a packet of the wrong version is received (#1197)

B.14. Since [draft-ietf-quic-transport-09](#)

- o 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)
- o A server can now only send 3 packets without validating the client address (#38, #1090)
- o Delivery order of stream data is no longer strongly specified (#252, #1070)
- o Rework of packet handling and version negotiation (#1038)
- o Stream 0 is now exempt from flow control until the handshake completes (#1074, #725, #825, #1082)
- o Improved retransmission rules for all frame types: information is retransmitted, not packets or frames (#463, #765, #1095, #1053)
- o Added an error code for server busy signals (#1137)
- o 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.15. Since [draft-ietf-quic-transport-08](#)

- o Clarified requirements for BLOCKED usage (#65, #924)

- o BLOCKED frame now includes reason for blocking (#452, #924, #927, #928)
- o GAP limitation in ACK Frame (#613)
- o Improved PMTUD description (#614, #1036)
- o Clarified stream state machine (#634, #662, #743, #894)
- o Reserved versions don't need to be generated deterministically (#831, #931)
- o You don't always need the draining period (#871)
- o Stateless reset clarified as version-specific (#930, #986)
- o `initial_max_stream_id_x` transport parameters are optional (#970, #971)
- o Ack Delay assumes a default value during the handshake (#1007, #1009)
- o Removed transport parameters from `NewSessionTicket` (#1015)

B.16. Since [draft-ietf-quic-transport-07](#)

- o The long header now has version before packet number (#926, #939)
- o Rename and consolidate packet types (#846, #822, #847)
- o Packet types are assigned new codepoints and the Connection ID Flag is inverted (#426, #956)
- o Removed type for Version Negotiation and use Version 0 (#963, #968)
- o 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)
- o The stream state machine has been split into read and write (#634, #894)

- o Employ variable-length integer encodings throughout (#595)
- o 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)
- o Address validation for connection migration (#161, #732, #878)
- o Clearly defined retransmission rules for BLOCKED (#452, #65, #924)
- o negotiated_version is sent in server transport parameters (#710, #959)
- o Increased the range over which packet numbers are randomized (#864, #850, #964)

B.17. Since [draft-ietf-quic-transport-06](#)

- o Replaced FNV-1a with AES-GCM for all "Cleartext" packets (#554)
- o Split error code space between application and transport (#485)
- o Stateless reset token moved to end (#820)
- o 1-RTT-protected long header types removed (#848)
- o No acknowledgments during draining period (#852)
- o Remove "application close" as a separate close type (#854)
- o Remove timestamps from the ACK frame (#841)
- o Require transport parameters to only appear once (#792)

B.18. Since [draft-ietf-quic-transport-05](#)

- o Stateless token is server-only (#726)
- o Refactor section on connection termination (#733, #748, #328, #177)
- o Limit size of Version Negotiation packet (#585)
- o Clarify when and what to ack (#736)

- o Renamed `STREAM_ID_NEEDED` to `STREAM_ID_BLOCKED`
- o Clarify Keep-alive requirements (#729)

B.19. Since [draft-ietf-quic-transport-04](#)

- o Introduce `STOP_SENDING` frame, `RESET_STREAM` only resets in one direction (#165)
- o Removed `GOAWAY`; application protocols are responsible for graceful shutdown (#696)
- o Reduced the number of error codes (#96, #177, #184, #211)
- o Version validation fields can't move or change (#121)
- o Removed versions from the transport parameters in a `NewSessionTicket` message (#547)
- o Clarify the meaning of "bytes in flight" (#550)
- o Public reset is now stateless reset and not visible to the path (#215)
- o Reordered bits and fields in `STREAM` frame (#620)
- o Clarifications to the stream state machine (#572, #571)
- o Increased the maximum length of the Largest Acknowledged field in `ACK` frames to 64 bits (#629)
- o `truncate_connection_id` is renamed to `omit_connection_id` (#659)
- o `CONNECTION_CLOSE` terminates the connection like TCP RST (#330, #328)
- o Update labels used in HKDF-Expand-Label to match TLS 1.3 (#642)

B.20. Since [draft-ietf-quic-transport-03](#)

- o Change `STREAM` and `RESET_STREAM` layout
- o Add `MAX_STREAM_ID` settings

B.21. Since [draft-ietf-quic-transport-02](#)

- o The size of the initial packet payload has a fixed minimum (#267, #472)
- o Define when Version Negotiation packets are ignored (#284, #294, #241, #143, #474)
- o The 64-bit FNV-1a algorithm is used for integrity protection of unprotected packets (#167, #480, #481, #517)
- o Rework initial packet types to change how the connection ID is chosen (#482, #442, #493)
- o No timestamps are forbidden in unprotected packets (#542, #429)
- o Cryptographic handshake is now on stream 0 (#456)
- o Remove congestion control exemption for cryptographic handshake (#248, #476)
- o Version 1 of QUIC uses TLS; a new version is needed to use a different handshake protocol (#516)
- o STREAM frames have a reduced number of offset lengths (#543, #430)
- o 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)
- o A NEW_CONNECTION_ID frame supports connection migration without linkability (#232, #491, #496)
- o 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)
- o Expanded security considerations (#440, #444, #445, #448)

B.22. Since [draft-ietf-quic-transport-01](#)

- o Defined short and long packet headers (#40, #148, #361)
- o Defined a versioning scheme and stable fields (#51, #361)
- o Define reserved version values for "greasing" negotiation (#112, #278)
- o The initial packet number is randomized (#35, #283)
- o Narrow the packet number encoding range requirement (#67, #286, #299, #323, #356)
- o Defined client address validation (#52, #118, #120, #275)
- o Define transport parameters as a TLS extension (#49, #122)
- o SCUP and COPT parameters are no longer valid (#116, #117)
- o Transport parameters for 0-RTT are either remembered from before, or assume default values (#126)
- o The server chooses connection IDs in its final flight (#119, #349, #361)
- o The server echoes the Connection ID and packet number fields when sending a Version Negotiation packet (#133, #295, #244)
- o Defined a minimum packet size for the initial handshake packet from the client (#69, #136, #139, #164)
- o Path MTU Discovery (#64, #106)
- o The initial handshake packet from the client needs to fit in a single packet (#338)
- o Forbid acknowledgment of packets containing only ACK and PADDING (#291)
- o Require that frames are processed when packets are acknowledged (#381, #341)
- o Removed the STOP_WAITING frame (#66)
- o Don't require retransmission of old timestamps for lost ACK frames (#308)

- o Clarified that frames are not retransmitted, but the information in them can be (#157, #298)
- o Error handling definitions (#335)
- o Split error codes into four sections (#74)
- o Forbid the use of Public Reset where CONNECTION_CLOSE is possible (#289)
- o Define packet protection rules (#336)
- o Require that stream be entirely delivered or reset, including acknowledgment of all STREAM frames or the RESET_STREAM, before it closes (#381)
- o Remove stream reservation from state machine (#174, #280)
- o Only stream 1 does not contribute to connection-level flow control (#204)
- o Stream 1 counts towards the maximum concurrent stream limit (#201, #282)
- o Remove connection-level flow control exclusion for some streams (except 1) (#246)
- o RESET_STREAM affects connection-level flow control (#162, #163)
- o Flow control accounting uses the maximum data offset on each stream, rather than bytes received (#378)
- o Moved length-determining fields to the start of STREAM and ACK (#168, #277)
- o Added the ability to pad between frames (#158, #276)
- o Remove error code and reason phrase from GOAWAY (#352, #355)
- o GOAWAY includes a final stream number for both directions (#347)
- o Error codes for RESET_STREAM and CONNECTION_CLOSE are now at a consistent offset (#249)
- o Defined priority as the responsibility of the application protocol (#104, #303)

B.23. Since [draft-ietf-quic-transport-00](#)

- o Replaced DIVERSIFICATION_NONCE flag with KEY_PHASE flag
- o Defined versioning
- o Reworked description of packet and frame layout
- o Error code space is divided into regions for each component
- o Use big endian for all numeric values

B.24. Since [draft-hamilton-quic-transport-protocol-01](#)

- o Adopted as base for [draft-ietf-quic-tls](#)
- o Updated authors/editors list
- o Added IANA Considerations section
- o 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: mt@lowentropy.net