QUIC
Internet-Draft
Intended status: Standards Track
Expires: February 16, 2018

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

August 15, 2017

QUIC: A UDP-Based Multiplexed and Secure Transport draft-ietf-quic-transport-05

Abstract

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

Note to Readers

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

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

Status of This Memo

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

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://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 February 16, 2018.

Table of Contents

1. Introduction 4
2. Conventions and Definitions 5
2.1. Notational Conventions 5
3. A QUIC Overview 6
3.1. Low-Latency Connection Establishment 6
3.2. Stream Multiplexing 6
3.3. Rich Signaling for Congestion Control and Loss Recovery . 7
3.4. Stream and Connection Flow Control 7
3.5. Authenticated and Encrypted Header and Payload 7
3.6. Connection Migration and Resilience to NAT Rebinding . . 8
3.7. Version Negotiation 8
4. Versions 8
5. Packet Types and Formats 9
5.1. Long Header 9
5.2. Short Header 11
5.3. Version Negotiation Packet 13
5.4. Cleartext Packets 14
5.4.1. Client Initial Packet 14
5.4.2. Server Stateless Retry Packet 15
5.4.3. Server Cleartext Packet 16
5.4.4. Client Cleartext Packet 16
5.5. Protected Packets 16
5.6. Connection ID 17
5.7. Packet Numbers 17
5.7.1. Initial Packet Number 19
5.8. Handling Packets from Different Versions 19
6. Frames and Frame Types 19
7. Life of a Connection 21
7.1. Version Negotiation 22
7.1.1. Using Reserved Versions 23
7.2. Cryptographic and Transport Handshake 23

7.3. Transport Parameters 24
7.3.1. Transport Parameter Definitions 26
7.3.2. Values of Transport Parameters for 0-RTT 27
7.3.3. New Transport Parameters 27
7.3.4. Version Negotiation Validation 28
7.4. Stateless Retries 29
7.5. Proof of Source Address Ownership 29
7.5.1. Client Address Validation Procedure 30
7.5.2. Address Validation on Session Resumption 31
7.5.3. Address Validation Token Integrity 32
7.6. Connection Migration 32
7.6.1. Privacy Implications of Connection Migration 32
7.6.2. Address Validation for Migrated Connections 33
7.7. Connection Termination 34
7.8. Stateless Reset 34
7.8.1. Detecting a Stateless Reset 36
7.8.2. Calculating a Stateless Reset Token 36
8. Frame Types and Formats 37
8.1. PADDING Frame 37
8.2. RST_STREAM Frame 37
8.3. CONNECTION_CLOSE frame 38
8.4. MAX_DATA Frame 39
8.5. MAX_STREAM_DATA Frame 39
8.6. MAX_STREAM_ID Frame 40
8.7. PING frame 41
8.8. BLOCKED Frame 41
8.9. STREAM_BLOCKED Frame 41
8.10. STREAM_ID_NEEDED Frame 42
8.11. NEW_CONNECTION_ID Frame 42
8.12. STOP_SENDING Frame 43
8.13. ACK Frame 43
8.13.1. ACK Block Section 46
8.13.2. Timestamp Section 46
8.13.3. ACK Frames and Packet Protection 48
8.14. STREAM Frame 49
9. Packetization and Reliability 51
9.1. Special Considerations for PMTU Discovery 53
10. Streams: QUIC's Data Structuring Abstraction 53
10.1. Stream Identifiers 54
10.2. Life of a Stream 54
10.2.1. idle 56
10.2.2. open 56
10.2.3. half-closed (local) 57
10.2.4. half-closed (remote) 57
10.2.5. closed 58
10.3. Solicited State Transitions 58
10.4. Stream Concurrency 59
10.5. Sending and Receiving Data 59

10.6. Stream Prioritization 60
11. Flow Control 61
11.1. Edge Cases and Other Considerations 62
11.1.1. Response to a RST_STREAM 63
11.1.2. Data Limit Increments 63
11.2. Stream Limit Increment 63
11.2.1. Blocking on Flow Control 64
11.3. Stream Final Offset 64
12. Error Handling 65
12.1. Connection Errors 65
12.2. Stream Errors 66
12.3. Error Codes 66
13. Security and Privacy Considerations 68
13.1. Spoofed ACK Attack 68
13.2. Slowloris Attacks 68
13.3. Stream Fragmentation and Reassembly Attacks 69
13.4. Stream Commitment Attack 69
14. IANA Considerations 70
14.1. QUIC Transport Parameter Registry 70
15. References 71
15.1. Normative References 71
15.2. Informative References 72
15.3. URIs 73
Appendix A. Contributors 73
Appendix B. Acknowledgments 73
Appendix C. Change Log 74
C.1. Since draft-ietf-quic-transport-04 74
C.2. Since draft-ietf-quic-transport-03 74
C.3. Since draft-ietf-quic-transport-02 75
C.4. Since draft-ietf-quic-transport-01 76
C.5. Since draft-ietf-quic-transport-00 78
C.6. Since draft-hamilton-quic-transport-protocol-01 78
Authors' Addresses 78

1. Introduction

QUIC is a multiplexed and secure transport protocol that runs on top of UDP. QUIC aims to provide a flexible set of features that allow it to be a general-purpose transport for multiple applications.

QUIC implements techniques learned from experience with TCP, SCTP and other transport protocols. QUIC uses UDP as substrate so as to not require changes to legacy client operating systems and middleboxes to be deployable. QUIC authenticates all of its headers and encrypts most of the data it exchanges, including its signaling. This allows the protocol to evolve without incurring a dependency on upgrades to middleboxes. This document describes the core QUIC protocol, including the conceptual design, wire format, and mechanisms of the

QUIC protocol for connection establishment, stream multiplexing, stream and connection-level flow control, and data reliability.

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

2. Conventions and Definitions

The words "MUST", "MUST NOT", "SHOULD", and "MAY" are used in this document. It's not shouting; when they are capitalized, they have the special meaning defined in [RFC2119].

Definitions of terms that are used in this document:

Client: The endpoint initiating a QUIC connection.

Server: The endpoint accepting incoming QUIC connections.

Endpoint: The client or server end of a connection.

Stream: A logical, bi-directional channel of ordered bytes within a QUIC connection.

Connection: A conversation between two QUIC endpoints with a single encryption context that multiplexes streams within it.

Connection ID: The identifier for a QUIC connection.

QUIC packet: A well-formed UDP payload that can be parsed by a QUIC receiver. QUIC packet size in this document refers to the UDP payload size.

2.1. Notational Conventions

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

[x] Indicates that x is optional

{x} Indicates that x is encrypted

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

3. A QUIC Overview

This section briefly describes QUIC's key mechanisms and benefits. Key strengths of QUIC include:

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

3.1. Low-Latency Connection Establishment

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

3.2. Stream Multiplexing

When application messages are transported over TCP, independent application messages can suffer from head-of-line blocking. When an application multiplexes many streams atop TCP's single-bytestream abstraction, a loss of a TCP segment results in blocking of all subsequent segments until a retransmission arrives, irrespective of the application streams that are encapsulated in subsequent segments. QUIC ensures that lost packets carrying data for an individual stream only impact that specific stream. Data received on other streams can continue to be reassembled and delivered to the application.

3.3. Rich Signaling for Congestion Control and Loss Recovery

QUIC's packet framing and acknowledgments carry rich information that help both congestion control and loss recovery in fundamental ways. Each QUIC packet carries a new packet number, including those carrying retransmitted data. This obviates the need for a separate mechanism to distinguish acknowledgments for retransmissions from those for original transmissions, avoiding TCP's retransmission ambiguity problem. QUIC acknowledgments also explicitly encode the delay between the receipt of a packet and its acknowledgment being sent, and together with the monotonically-increasing packet numbers, this allows for precise network roundtrip-time (RTT) calculation. QUIC's ACK frames support up to 256 ACK blocks, so QUIC is more resilient to reordering than TCP with SACK support, as well as able to keep more bytes on the wire when there is reordering or loss.

3.4. Stream and Connection Flow Control

QUIC implements stream- and connection-level flow control. At a high level, a QUIC receiver advertises the maximum amount of data that it is willing to receive on each stream. As data is sent, received, and delivered on a particular stream, the receiver sends MAX_STREAM_DATA frames that increase the advertised limit for that stream, allowing the peer to send more data on that stream.

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

3.5. Authenticated and Encrypted Header and Payload

TCP headers appear in plaintext on the wire and are not authenticated, causing a plethora of injection and header manipulation issues for TCP, such as receive-window manipulation and sequence-number overwriting. While some of these are mechanisms used by middleboxes to improve TCP performance, others are active attacks. Even "performance-enhancing" middleboxes that routinely interpose on the transport state machine end up limiting the evolvability of the transport protocol, as has been observed in the design of MPTCP [RFC6824] and in its subsequent deployability issues.

Generally, QUIC packets are always authenticated and the payload is typically fully encrypted. The parts of the packet header which are not encrypted are still authenticated by the receiver, so as to thwart any packet injection or manipulation by third parties. Some

early handshake packets, such as the Version Negotiation packet, are not encrypted, but information sent in these unencrypted handshake packets is later verified as part of cryptographic processing.

3.6. Connection Migration and Resilience to NAT Rebinding

QUIC connections are identified by a 64-bit Connection ID, randomly generated by the server. QUIC's consistent connection ID allows connections to survive changes to the client's IP and port, such as those caused by NAT rebindings or by the client changing network connectivity to a new address. QUIC provides automatic cryptographic verification of a rebound client, since the client continues to use the same session key for encrypting and decrypting packets. The consistent connection ID can be used to allow migration of the connection to a new server IP address as well, since the Connection ID remains consistent across changes in the client's and the server's network addresses.

3.7. Version Negotiation

QUIC version negotiation allows for multiple versions of the protocol to be deployed and used concurrently. Version negotiation is described in Section 7.1.

4. Versions

QUIC versions are identified using a 32-bit value.

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

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

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

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

Reserved version numbers will probably never represent a real protocol; a client MAY use one of these version numbers with the expectation that the server will initiate version negotiation; a

server MAY advertise support for one of these versions and can expect that clients ignore the value.

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

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

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

Implementors are encouraged to register version numbers of QUIC that they are using for private experimentation on the github wiki [4].

5. Packet Types and Formats

We first describe QUIC's packet types and their formats, since some are referenced in subsequent mechanisms.

All numeric values are encoded in network byte order (that is, big-endian) and all field sizes are in bits. When discussing individual bits of fields, the least significant bit is referred to as bit 0. Hexadecimal notation is used for describing the value of fields.

Any QUIC packet has either a long or a short header, as indicated by the Header Form bit. Long headers are expected to be used early in the connection before version negotiation and establishment of 1-RTT keys. Short headers are minimal version-specific headers, which can be used after version negotiation and 1-RTT keys are established.

5.1. Long Header

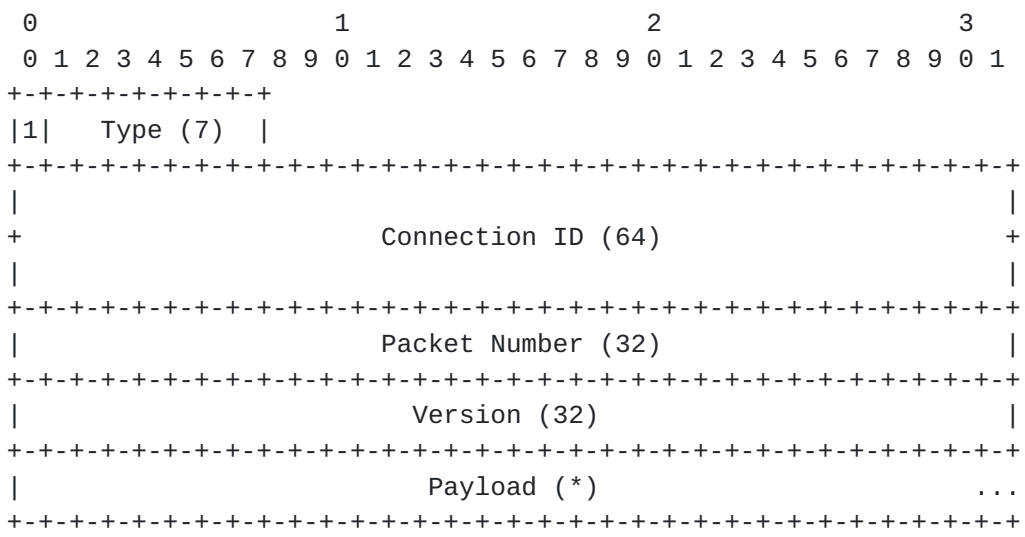


Figure 1: Long Header Format

Long headers are used for packets that are sent prior to the completion of version negotiation and establishment of 1-RTT keys. Once both conditions are met, a sender SHOULD switch to sending short-form headers. While inefficient, long headers MAY be used for packets encrypted with 1-RTT keys. The long form allows for special packets - such as the Version Negotiation packet - to be represented in this uniform fixed-length packet format. A long header contains the following fields:

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

Long Packet Type: The remaining seven bits of octet 0 contain the packet type. This field can indicate one of 128 packet types. The types specified for this version are listed in Table 1.

Connection ID: Octets 1 through 8 contain the connection ID. Section 5.6 describes the use of this field in more detail.

Packet Number: Octets 9 to 12 contain the packet number. Section 5.7 describes the use of packet numbers.

Version: Octets 13 to 16 contain the selected protocol version. This field indicates which version of QUIC is in use and determines how the rest of the protocol fields are interpreted.

Payload: Octets from 17 onwards (the rest of QUIC packet) are the payload of the packet.

The following packet types are defined:

Type	Name	Section
0x01	Version Negotiation	Section 5.3
0x02	Client Initial	Section 5.4.1
0x03	Server Stateless Retry	Section 5.4.2
0x04	Server Cleartext	Section 5.4.3
0x05	Client Cleartext	Section 5.4.4
0x06	0-RTT Protected	Section 5.5
0x07	1-RTT Protected (key phase 0)	Section 5.5
0x08	1-RTT Protected (key phase 1)	Section 5.5

Table 1: Long Header Packet Types

The header form, packet type, connection ID, packet number and version fields of a long header packet are version-independent. The types of packets defined in Table 1 are version-specific. See Section 5.8 for details on how packets from different versions of QUIC are interpreted.

(TODO: Should the list of packet types be version-independent?)

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

5.2. Short Header

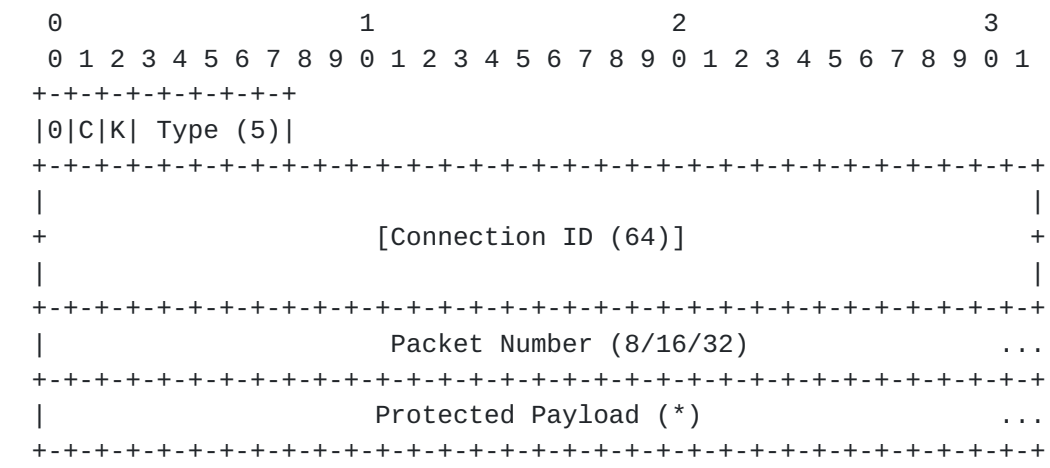


Figure 2: Short Header Format

The short header can be used after the version and 1-RTT keys are negotiated. This header form has the following fields:

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

Connection ID Flag: The second bit (0x40) of octet 0 indicates whether the Connection ID field is present. If set to 1, then the Connection ID field is present; if set to 0, the Connection ID field is omitted. The Connection ID field can only be omitted if the omit_connection_id transport parameter (Section 7.3.1) is specified by the intended recipient of the packet.

Key Phase Bit: The third bit (0x20) of octet 0 indicates the key phase, which allows a recipient of a packet to identify the packet protection keys that are used to protect the packet. See [QUIC-TLS] for details.

Short Packet Type: The remaining 5 bits of octet 0 include one of 32 packet types. Table 2 lists the types that are defined for short packets.

Connection ID: If the Connection ID Flag is set, a connection ID occupies octets 1 through 8 of the packet. See Section 5.6 for more details.

Packet Number: The length of the packet number field depends on the packet type. This field can be 1, 2 or 4 octets long depending on the short packet type.

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

The packet type in a short header currently determines only the size of the packet number field. Additional types can be used to signal the presence of other fields.

Type	Packet Number Size
0x01	1 octet
0x02	2 octets
0x03	4 octets

Table 2: Short Header Packet Types

The header form, connection ID flag and connection ID of a short header packet are version-independent. The remaining fields are specific to the selected QUIC version. See Section 5.8 for details on how packets from different versions of QUIC are interpreted.

5.3. Version Negotiation Packet

A Version Negotiation packet has long headers with a type value of 0x01 and is sent only by servers. The Version Negotiation packet is a response to a client packet that contains a version that is not supported by the server.

The packet number, connection ID and version fields echo corresponding values from the triggering client packet. This allows clients some assurance that the server received the packet and that the Version Negotiation packet was not carried in a packet with a spoofed source address.

A Version Negotiation packet is never explicitly acknowledged in an ACK frame by a client. Receiving another Client Initial packet implicitly acknowledges a Version Negotiation packet.

The payload of the Version Negotiation packet is a list of 32-bit versions which the server supports, as shown below.

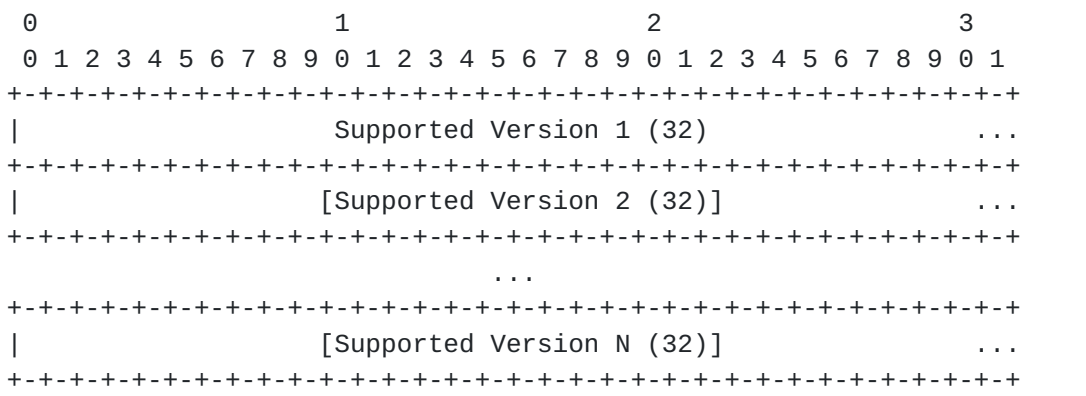


Figure 3: Version Negotiation Packet

See Section 7.1 for a description of the version negotiation process.

5.4. Cleartext Packets

Cleartext packets are sent during the handshake prior to key negotiation.

All cleartext packets contain the current QUIC version in the version field.

The payload of cleartext packets also includes an integrity check, which is described in [QUIC-TLS].

5.4.1. Client Initial Packet

The Client Initial packet uses long headers with a type value of 0x02. It carries the first cryptographic handshake message sent by the client.

The client populates the connection ID field with randomly selected values, unless it has received a packet from the server. If the client has received a packet from the server, the connection ID field uses the value provided by the server.

The packet number used for Client Initial packets is initialized with a random value each time the new contents are created for the packet. Retransmissions of the packet contents increment the packet number by one, see (Section 5.7).

The payload of a Client Initial packet consists of a STREAM frame (or frames) for stream 0 containing a cryptographic handshake message, with enough PADDING frames that the packet is at least 1200 octets (see Section 9). The stream in this packet always starts at an

offset of 0 (see Section 7.4) and the complete cyptographic handshake message MUST fit in a single packet (see Section 7.2).

The client uses the Client 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, this includes the packets sent after receiving a Version Negotiation (Section 5.3) or Server Stateless Retry packet (Section 5.4.2).

5.4.2. Server Stateless Retry Packet

A Server Stateless Retry packet uses long headers with a type value of 0x03. It carries cryptographic handshake messages and acknowledgments. It is used by a server that wishes to perform a stateless retry (see Section 7.4).

The packet number and connection ID fields echo the corresponding fields from the triggering client packet. This allows a client to verify that the server received its packet.

A Server Stateless Retry packet is never explicitly acknowledged in an ACK frame by a client. Receiving another Client Initial packet implicitly acknowledges a Server Stateless Retry packet.

After receiving a Server Stateless Retry packet, the client uses a new Client Initial packet containing the next cryptographic handshake message. The client retains the state of its cryptographic handshake, but discards all transport state. In effect, the next cryptographic handshake message is sent on a new connection. The new Client Initial packet is sent in a packet with a newly randomized packet number and starting at a stream offset of 0.

Continuing the cryptographic handshake is necessary to ensure that an attacker cannot force a downgrade of any cryptographic parameters. In addition to continuing the cryptographic handshake, the client MUST remember the results of any version negotiation that occurred (see Section 7.1). The client MAY also retain any observed RTT or congestion state that it has accumulated for the flow, but other transport state MUST be discarded.

The payload of the Server Stateless Retry packet contains STREAM frames and could contain PADDING and ACK frames. A server can only send a single Server Stateless Retry packet in response to each Client Initial packet that is receives.

5.4.3. Server Cleartext Packet

A Server Cleartext packet uses long headers with a type value of 0x04. It is used to carry acknowledgments and cryptographic handshake messages from the server.

The connection ID field in a Server Cleartext packet contains a connection ID that is chosen by the server (see Section 5.6).

The first Server Cleartext packet contains a randomized packet number. This value is increased for each subsequent packet sent by the server as described in Section 5.7.

The payload of this packet contains STREAM frames and could contain PADDING and ACK frames.

5.4.4. Client Cleartext Packet

A Client Cleartext packet uses long headers with a type value of 0x05, and is sent when the client has received a Server Cleartext packet from the server.

The connection ID field in a Client Cleartext packet contains a server-selected connection ID, see Section 5.6.

The Client Cleartext packet includes a packet number that is one higher than the last Client Initial, 0-RTT Protected or Client Cleartext packet that was sent. The packet number is incremented for each subsequent packet, see Section 5.7.

The payload of this packet contains STREAM frames and could contain PADDING and ACK frames.

5.5. Protected Packets

Packets that are protected with 0-RTT keys are sent with long headers. Packets that are protected with 1-RTT keys MAY be sent with long headers. The different packet types explicitly indicate the encryption level and therefore the keys that are used to remove packet protection.

Packets protected with 0-RTT keys use a type value of 0x06. The connection ID field for a 0-RTT packet is selected by the client.

The client can send 0-RTT packets after having received a packet from the server if that packet does not complete the handshake. Even if the client receives a different connection ID from the server, it

MUST NOT update the connection ID it uses for 0-RTT packets. This enables consistent routing for all 0-RTT packets.

Packets protected with 1-RTT keys that use long headers use a type value of 0x07 for key phase 0 and 0x08 for key phase 1; see [QUIC-TLS] for more details on the use of key phases. The connection ID field for these packet types MUST contain the value selected by the server, see Section 5.6.

The version field for protected packets is the current QUIC version.

The packet number field contains a packet number, which increases with each packet sent, see Section 5.7 for details.

The payload is protected using authenticated encryption. [QUIC-TLS] describes packet protection in detail. After decryption, the plaintext consists of a sequence of frames, as described in Section 6.

5.6. Connection ID

QUIC connections are identified by their 64-bit Connection ID. All long headers contain a Connection ID. Short headers indicate the presence of a Connection ID using the CONNECTION_ID flag. When present, the Connection ID is in the same location in all packet headers, making it straightforward for middleboxes, such as load balancers, to locate and use it.

The client MUST choose a random connection ID and use it in Client Initial packets (Section 5.4.1) and 0-RTT packets (Section 5.5). If the client has received any packet from the server, it uses the connection ID it received from the server for all packets other than 0-RTT packets.

When the server receives a Client Initial packet and decides to proceed with the handshake, it chooses a new value for the connection ID and sends that in a Server Cleartext packet. The server MAY choose to use the value that the client initially selects.

Once the client receives the connection ID that the server has chosen, it uses this for all subsequent packets that it sends, except for any 0-RTT packets, which all have the same connection ID.

5.7. Packet Numbers

The packet number is a 64-bit unsigned number and is used as part of a cryptographic nonce for packet encryption. Each endpoint maintains a separate packet number for sending and receiving. The packet

number for sending MUST increase by at least one after sending any packet, unless otherwise specified (see Section 5.7.1).

A QUIC endpoint MUST NOT reuse a packet number within the same connection (that is, under the same cryptographic keys). If the packet number for sending reaches 2^64 - 1, the sender MUST close the connection without sending a CONNECTION_CLOSE frame or any further packets; the sender MAY send a Public Reset packet in response to further packets that it receives.

To reduce the number of bits required to represent the packet number over the wire, only the least significant bits of the packet number are transmitted. The actual packet number for each packet is reconstructed at the receiver based on the largest packet number received on a successfully authenticated packet.

A packet number is decoded by finding the packet number value that is closest to the next expected packet. The next expected packet is the highest received packet number plus one. For example, if the highest successfully authenticated packet had a packet number of 0xaa82f30e, then a packet containing a 16-bit value of 0x1f94 will be decoded as 0xaa831f94.

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

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

For example, if an endpoint has received an acknowledgment for packet 0x6afa2f, sending a packet with a number of 0x6b4264 requires a 16-bit or larger packet number encoding; whereas a 32-bit packet number is needed to send a packet with a number of 0x6bc107.

Version Negotiation (Section 5.3) and Server Stateless Retry (Section 5.4.2) packets have special rules for populating the packet number field.

5.7.1. Initial Packet Number

The initial value for packet number MUST be selected from an uniform random distribution between 0 and 2^31-1. That is, the lower 31 bits of the packet number are randomized. [RFC4086] provides guidance on the generation of random values.

The first set of packets sent by an endpoint MUST include the low 32-bits of the packet number. Once any packet has been acknowledged, subsequent packets can use a shorter packet number encoding.

A client that receives a Version Negotiation (Section 5.3) or Server Stateless Retry packet (Section 5.4.2) MUST generate a new initial packet number. This ensures that the first transmission attempt for a Client Initial packet (Section 5.4.1) always contains a randomized packet number, but packets that contain retransmissions increment the packet number.

A client MUST NOT generate a new initial packet number if it discards the server packet. This might happen if the information the client retransmits its Client Initial packet.

5.8. Handling Packets from Different Versions

Between different versions the following things are guaranteed to remain constant:

- the location of the header form flag,
- the location of the Connection ID flag in short headers,
- the location and size of the Connection ID field in both header forms,
- the location and size of the Version field in long headers, and
- the location and size of the Packet Number field in long headers.

Implementations MUST assume that an unsupported version uses an unknown packet format. All other fields MUST be ignored when processing a packet that contains an unsupported version.

6. Frames and Frame Types

The payload of cleartext packets and the plaintext after decryption of protected payloads consists of a sequence of frames, as shown in Figure 4.

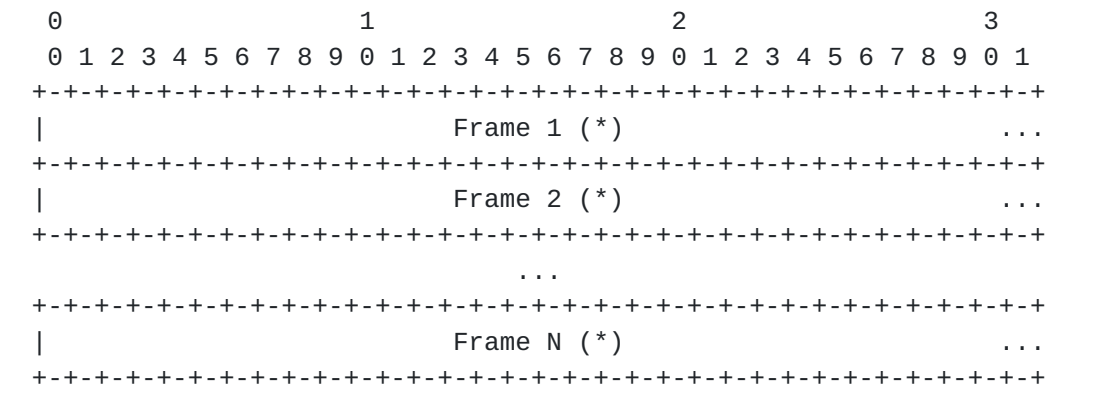


Figure 4: Contents of Protected Payload

Protected payloads MUST contain at least one frame, and MAY contain multiple frames and multiple frame types.

Frames MUST fit within a single QUIC packet and MUST NOT span a QUIC packet boundary. Each frame begins with a Frame Type byte, indicating its type, followed by additional type-dependent fields:

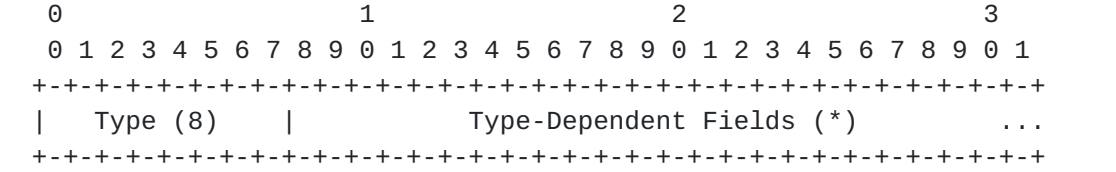


Figure 5: Generic Frame Layout

Frame types are listed in Table 3. Note that the Frame Type byte in STREAM and ACK frames is used to carry other frame-specific flags. For all other frames, the Frame Type byte simply identifies the frame. These frames are explained in more detail as they are referenced later in the document.

Type Value	Frame Type Name	Definition
0x00	PADDING	Section 8.1
0x01	RST_STREAM	Section 8.2
0x02	CONNECTION_CLOSE	Section 8.3
0x04	MAX_DATA	Section 8.4
0x05	MAX_STREAM_DATA	Section 8.5
0x06	MAX_STREAM_ID	Section 8.6
0x07	PING	Section 8.7
0x08	BLOCKED	Section 8.8
0x09	STREAM_BLOCKED	Section 8.9
0x0a	STREAM_ID_NEEDED	Section 8.10
0x0b	NEW_CONNECTION_ID	Section 8.11
0x0c	STOP_SENDING	Section 8.12
0xa0 - 0xbf	ACK	Section 8.13
0xc0 - 0xff	STREAM	Section 8.14

Table 3: Frame Types

7. Life of a Connection

A QUIC connection is a single conversation between two QUIC endpoints. QUIC's connection establishment intertwines version negotiation with the cryptographic and transport handshakes to reduce connection establishment latency, as described in Section 7.2. Once established, a connection may migrate to a different IP or port at either endpoint, due to NAT rebinding or mobility, as described in Section 7.6. Finally a connection may be terminated by either endpoint, as described in Section 7.7.

7.1. Version Negotiation

QUIC's connection establishment begins with version negotiation, since all communication between the endpoints, including packet and frame formats, relies on the two endpoints agreeing on a version.

A QUIC connection begins with a client sending a handshake packet. The details of the handshake mechanisms are described in Section 7.2, but all of the initial packets sent from the client to the server MUST use the long header format and MUST specify the version of the protocol being used.

When the server receives a packet from a client with the long header format, it compares the client's version to the versions it supports.

If the version selected by the client is not acceptable to the server, the server discards the incoming packet and responds with a Version Negotiation packet (Section 5.3). This includes a list of versions that the server will accept.

A server sends a Version Negotiation packet for every packet that it receives with an unacceptable version. This allows a server to process packets with unsupported versions without retaining state. Though either the initial client 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.

If the packet contains a version that is acceptable to the server, the server proceeds with the handshake (Section 7.2). This commits the server to the version that the client selected.

When the client receives a Version Negotiation packet from the server, it should select an acceptable protocol version. If the server lists an acceptable version, the client selects that version and reattempts to create a connection using that version. Though the contents of a packet might not change in response to version negotiation, a client MUST increase the packet number it uses on every packet it sends. Packets MUST continue to use long headers and MUST include the new negotiated protocol version.

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

A client MUST NOT change the version it uses unless it is in response to a Version Negotiation packet from the server. Once a client receives a packet from the server which is not a Version Negotiation packet, it MUST ignore other Version Negotiation packets on the same

connection. Similarly, a client MUST ignore a Version Negotiation packet if it has already received and acted on a Version Negotiation packet.

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

Version negotiation uses unprotected data. The result of the negotiation MUST be revalidated as part of the cryptographic handshake (see Section 7.3.4).

7.1.1. Using Reserved Versions

For a server to use a new version in the future, clients must correctly handle unsupported versions. To help ensure this, a server SHOULD include a reserved version (see Section 4) while generating a Version Negotiation packet.

The design of version negotiation permits a server to avoid maintaining state for packets that it rejects in this fashion. However, when the server generates a Version Negotiation packet, it cannot randomly generate a reserved version number. This is because the server is required to include the same value in its transport parameters (see Section 7.3.4). To avoid the selected version number changing during connection establishment, the reserved version SHOULD be generated as a function of values that will be available to the server when later generating its handshake packets.

A pseudorandom function that takes client address information (IP and port) and the client selected version as input would ensure that there is sufficient variability in the values that a server uses.

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

7.2. Cryptographic and Transport Handshake

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

QUIC provides this stream with reliable, ordered delivery of data. In return, the cryptographic handshake provides QUIC with:

- authenticated key exchange, where

    * a server is always authenticated,

    * a client is optionally authenticated,

    * every connection produces distinct and unrelated keys,

    * keying material is usable for packet protection for both 0-RTT and 1-RTT packets, and

    * 1-RTT keys have forward secrecy

- authenticated values for the transport parameters of the peer (see Section 7.3)

- authenticated confirmation of version negotiation (see Section 7.3.4)

- authenticated negotiation of an application protocol (TLS uses ALPN [RFC7301] for this purpose)

- for the server, the ability to carry data that provides assurance that the client can receive packets that are addressed with the transport address that is claimed by the client (see Section 7.5)

The initial cryptographic handshake message MUST be sent in a single packet. Any second attempt that is triggered by address validation MUST also be sent within a single packet. This avoids having to reassemble a message from multiple packets. Reassembling messages requires that a server maintain state prior to establishing a connection, exposing the server to a denial of service risk.

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

Details of how TLS is integrated with QUIC is provided in more detail in [QUIC-TLS].

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 format of the transport parameters is the TransportParameters struct from Figure 6. This is described using the presentation language from Section 3 of [I-D.ietf-tls-tls13].

```
uint32 QuicVersion;

enum {
   initial_max_stream_data(0),
   initial_max_data(1),
   initial_max_stream_id(2),
   idle_timeout(3),
   omit_connection_id(4),
   max_packet_size(5),
   stateless_reset_token(6),
   (65535)
} TransportParameterId;

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

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

      case encrypted_extensions:
         QuicVersion supported_versions<2..2^8-4>;

      case new_session_ticket:
         struct {};
   };
   TransportParameter parameters<30..2^16-1>;
} TransportParameters;
```

Figure 6: 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 encode the transport parameters.

QUIC encodes transport parameters into a sequence of octets, which are then included in the cryptographic handshake. Once the handshake completes, the transport parameters declared by the peer are available. Each endpoint validates the value provided by its peer. In particular, version negotiation MUST be validated (see

Section 7.3.4) before the connection establishment is considered properly complete.

Definitions for each of the defined transport parameters are included in Section 7.3.1.

7.3.1. Transport Parameter Definitions

An endpoint MUST include the following parameters in its encoded TransportParameters:

initial_max_stream_data (0x0000): The initial stream maximum data parameter contains the initial value for the maximum data that can be sent on any newly created stream. This parameter is encoded as an unsigned 32-bit integer in units of octets. This is equivalent to an implicit MAX_STREAM_DATA frame (Section 8.5) being sent on all streams immediately after opening.

initial_max_data (0x0001): The initial maximum data parameter contains the initial value for the maximum amount of data that can be sent on the connection. This parameter is encoded as an unsigned 32-bit integer in units of 1024 octets. That is, the value here is multiplied by 1024 to determine the actual maximum value. This is equivalent to sending a MAX_DATA (Section 8.4) for the connection immediately after completing the handshake.

initial_max_stream_id (0x0002): The initial maximum stream ID parameter contains the initial maximum stream number the peer may initiate, encoded as an unsigned 32-bit integer. This is equivalent to sending a MAX_STREAM_ID (Section 8.6) immediately after completing the handshake.

idle_timeout (0x0003): The idle timeout is a value in seconds that is encoded as an unsigned 16-bit integer. The maximum value is 600 seconds (10 minutes).

stateless_reset_token (0x0005): The Stateless Reset Token is used in verifying a stateless reset, see Section 7.8. This parameter is a sequence of 16 octets.

An endpoint MAY use the following transport parameters:

omit_connection_id (0x0004): The omit connection identifier parameter indicates that packets sent to the endpoint that advertises this parameter can omit the connection ID. This can be used by an endpoint where it knows that source and destination IP address and port are sufficient for it to identify a connection. This parameter is zero length. Absence this parameter indicates

that the endpoint relies on the connection ID being present in every packet.

max_packet_size (0x0005): The maximum packet size parameter places a limit on the size of packets that the endpoint is willing to receive, encoded as an unsigned 16-bit integer. This indicates that packets larger than this limit will be dropped. The default for this parameter is the maximum permitted UDP payload of 65527. Values below 1200 are invalid. This limit only applies to protected packets (Section 5.5).

7.3.2. Values of Transport Parameters for 0-RTT

Transport parameters from the server MUST be remembered by the client for use with 0-RTT data. If the TLS NewSessionTicket message includes the quic_transport_parameters extension, then those values are used for the server values when establishing a new connection using that ticket. Otherwise, the transport parameters that the server advertises during connection establishment are used.

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

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

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

7.3.3. New Transport Parameters

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

New transport parameters can be registered according to the rules in Section 14.1.

7.3.4. Version Negotiation Validation

The transport parameters include three fields that encode version information. These retroactively authenticate the version negotiation (see Section 7.1) that is performed prior to the cryptographic handshake.

The cryptographic handshake provides integrity protection for the negotiated version as part of the transport parameters (see Section 7.3). As a result, modification of version negotiation packets by an attacker can be detected.

The client includes two fields in the transport parameters:

- The negotiated_version is the version that was finally selected for use. This MUST be identical to the value that is on the packet that carries the ClientHello. A server that receives a negotiated_version that does not match the version of QUIC that is in use MUST terminate the connection with a VERSION_NEGOTIATION_ERROR error code.

- The initial_version is the version that the client initially attempted to use. If the server did not send a version negotiation packet Section 5.3, this will be identical to the negotiated_version.

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

A server that does not maintain state for every packet it receives (i.e., a stateless server) uses a different process. If the initial and negotiated versions are the same, a stateless server can accept the value.

If the initial version is different from the negotiated_version, a stateless server MUST check that it would have sent a version negotiation packet if it had received a packet with the indicated initial_version. If a server would have accepted the version included in the initial_version and the value differs from the value of negotiated_version, the server MUST terminate the connection with a VERSION_NEGOTIATION_ERROR error.

The server includes a list of versions that it would send in any version negotiation packet (Section 5.3) in supported_versions. The server populates this field even if it did not send a version negotiation packet. This field is absent if the parameters are included in a NewSessionTicket message.

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

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

7.4. Stateless Retries

A server can process an initial cryptographic handshake messages from a client without committing any state. This allows a server to perform address validation (Section 7.5, or to defer connection establishment costs.

A server that generates a response to an initial packet without retaining connection state MUST use the Server Stateless Retry packet (Section 5.4.2). This packet causes a client to reset its transport state and to continue the connection attempt with new connection state while maintaining the state of the cryptographic handshake.

A server MUST NOT send multiple Server Stateless Retry packets in response to a client handshake packet. Thus, any cryptographic handshake message that is sent MUST fit within a single packet.

In TLS, the Server Stateless Retry packet type is used to carry the HelloRetryRequest message.

7.5. Proof of Source Address Ownership

Transport protocols commonly spend a round trip checking that a client owns the transport address (IP and port) that it claims. Verifying that a client can receive packets sent to its claimed transport address protects against spoofing of this information by malicious clients.

This technique is used primarily to avoid QUIC from being used for 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.

Several methods are used in QUIC to mitigate this attack. Firstly, the initial handshake packet is padded to at least 1280 octets. This allows a server to send a similar amount of data without risking causing an amplification attack toward an unproven remote address.

A server eventually confirms that a client has received its messages when the cryptographic handshake successfully completes. This might be insufficient, either because the server wishes to avoid the computational cost of completing the handshake, or it might be that the size of the packets that are sent during the handshake is too large. This is especially important for 0-RTT, where the server might wish to provide application data traffic - such as a response to a request - in response to the data carried in the early data from the client.

To send additional data prior to completing the cryptographic handshake, the server then needs to validate that the client owns the address that it claims.

Source address validation is therefore performed during the establishment of a connection. TLS provides the tools that support the feature, but basic validation is performed by the core transport protocol.

7.5.1. Client Address Validation Procedure

QUIC uses token-based address validation. Any time the server wishes to validate a client address, it provides the client with a token. As long as the token cannot be easily guessed (see Section 7.5.3), if the client is able to return that token, it proves to the server that it received the token.

During the processing of the cryptographic handshake messages from a client, TLS will request that QUIC make a decision about whether to proceed based on the information it has. TLS will provide QUIC with any token that was provided by the client. For an initial packet, QUIC can decide to abort the connection, allow it to proceed, or request address validation.

If QUIC decides to request address validation, it provides the cryptographic handshake with a token. The contents of this token are

consumed by the server that generates the token, so there is no need for a single well-defined format. 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.

The cryptographic handshake is responsible for enacting validation by sending the address validation token to the client. A legitimate client will include a copy of the token when it attempts to continue the handshake. The cryptographic handshake extracts the token then asks QUIC a second time whether the token is acceptable. In response, QUIC can either abort the connection or permit it to proceed.

A connection MAY be accepted without address validation - or with only limited validation - but a server SHOULD limit the data it sends toward an unvalidated address. Successful completion of the cryptographic handshake implicitly provides proof that the client has received packets from the server.

7.5.2. Address Validation on Session Resumption

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.

A different type of token is needed when resuming. Unlike the token that is created during a handshake, there might be some time between when the token is created and when the token is subsequently used. Thus, a resumption token SHOULD include an expiration time. It is also unlikely that the client port number is the same on two different connections; validating the port is therefore unlikely to be successful.

This token can be provided to the cryptographic handshake immediately after establishing a connection. QUIC might also generate an updated token if significant time passes or the client address changes for any reason (see Section 7.6). The cryptographic handshake is responsible for providing the client with the token. In TLS the token is included in the ticket that is used for resumption and 0-RTT, which is carried in a NewSessionTicket message.

7.5.3. 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.

In TLS the address validation token is often bundled with the information that TLS requires, such as the resumption secret. In this case, adding integrity protection can be delegated to the cryptographic handshake protocol, avoiding redundant protection. If integrity protection is delegated to the cryptographic handshake, an integrity failure will result in immediate cryptographic handshake failure. If integrity protection is performed by QUIC, QUIC MUST abort the connection if the integrity check fails with a PROTOCOL_VIOLATION error code.

7.6. Connection Migration

QUIC connections are identified by their 64-bit Connection ID. QUIC's consistent connection ID allows connections to survive changes to the client's IP and/or port, such as those caused by client or server migrating to a new network. Connection migration allows a client to retain any shared state with a connection when they move networks. This includes state that can be hard to recover such as outstanding requests, which might otherwise be lost with no easy way to retry them.

7.6.1. Privacy Implications of Connection Migration

Using a stable connection ID on multiple network paths allows a passive observer to correlate activity between those paths. A client that moves between networks might not wish to have their activity correlated by any entity other than a server. The NEW_CONNECTION_ID message can be sent by a server to provide an unlinkable connection ID for use in case the client wishes to explicitly break linkability between two points of network attachment.

A client might need to send packets on multiple networks without receiving any response from the server. To ensure that the client is not linkable across each of these changes, a new connection ID and

packet number gap are needed for each network. To support this, a server sends multiple NEW_CONNECTION_ID messages. Each NEW_CONNECTION_ID is marked with a sequence number. Connection IDs MUST be used in the order in which they are numbered.

A client which wishes to break linkability upon changing networks MUST use the connection ID provided by the server as well as incrementing the packet sequence number by an externally unpredictable value computed as described in Section 7.6.1.1. Packet number gaps are cumulative. A client might skip connection IDs, but it MUST ensure that it applies the associated packet number gaps for connection IDs that it skips in addition to the packet number gap associated with the connection ID that it does use.

A server that receives a packet that is marked with a new connection ID recovers the packet number by adding the cumulative packet number gap to its expected packet number. A server SHOULD discard packets that contain a smaller gap than it advertised.

For instance, a server might provide a packet number gap of 7 associated with a new connection ID. If the server received packet 10 using the previous connection ID, it should expect packets on the new connection ID to start at 18. A packet with the new connection ID and a packet number of 17 is discarded as being in error.

7.6.1.1. Packet Number Gap

In order to avoid linkage, the packet number gap MUST be externally indistinguishable from random. The packet number gap for a connection ID with sequence number is computed by encoding the sequence number as a 32-bit integer in big-endian format, and then computing:

```
Gap = HKDF-Expand-Label(packet_number_secret,
                        "QUIC packet sequence gap", sequence, 4)
```

The output of HKDF-Expand-Label is interpreted as a big-endian number. "packet_number_secret" is derived from the TLS key exchange, as described in [QUIC-TLS] Section 5.6.

7.6.2. Address Validation for Migrated Connections

TODO: see issue #161

7.7. Connection Termination

Connections should remain open until they become idle for a pre-negotiated period of time. A QUIC connection, once established, can be terminated in one of three ways:

1. Explicit Shutdown: An endpoint sends a CONNECTION_CLOSE frame to terminate the connection. An endpoint MAY use application-layer mechanisms prior to a CONNECTION_CLOSE to indicate that the connection will soon be terminated. On termination of the active streams, a CONNECTION_CLOSE may be sent. If an endpoint sends a CONNECTION_CLOSE frame while unterminated streams are active (no FIN bit or RST_STREAM frames have been sent or received for one or more streams), then the peer must assume that the streams were incomplete and were abnormally terminated.

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

3. Stateless Reset: An endpoint that loses state can use this procedure to cause the connection to terminate early, see Section 7.8 for details.

After receiving either a CONNECTION_CLOSE frame or a Public Reset, an endpoint MUST NOT send additional packets on that connection. After sending either a CONNECTION_CLOSE frame or a Public Reset packet, implementations MUST NOT send any non-closing packets on that connection. If additional packets are received after this time and before idle_timeout seconds has passed, implementations SHOULD respond to them by sending a CONNECTION_CLOSE (which MAY just be a duplicate of the previous CONNECTION_CLOSE packet) but MAY also send a Public Reset packet. Implementations SHOULD throttle these responses, for instance by exponentially backing off the number of packets which are received before sending a response. After this time, implementations SHOULD respond to unexpected packets with a Public Reset packet.

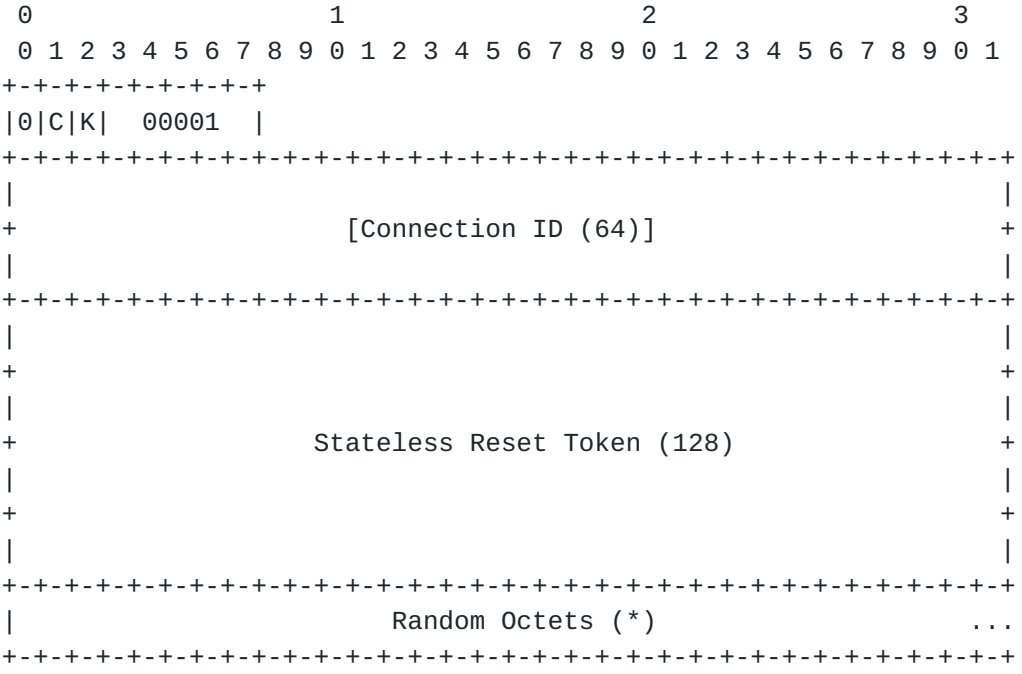
7.8. Stateless Reset

A stateless reset is provided as an option of last resort for a server that does not have access to the state of a connection. A server crash or outage might result in clients continuing to send data to a server that is unable to properly continue the connection.

A server 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, the server sends a stateless_reset_token value during the handshake in the transport parameters. This value is protected by encryption, so only client and server know this value.

A server that receives packets that it cannot process sends a packet in the following layout:



This packet SHOULD use the short header form with the shortest possible packet number encoding. This minimizes the perceived gap between the last packet that the server sent and this packet. The leading octet of the Stateless Reset Token will be interpreted as a packet number. A server MAY use a different short header type, indicating a different packet number length, but this allows for the message to be identified as a stateless reset more easily using heuristics.

A server copies the connection ID field from the packet that triggers the stateless reset. A server omits the connection ID if explicitly configured to do so, or if the client packet did not include a connection ID.

After the first short header octet and optional connection ID, the server includes the value of the Stateless Reset Token that it included in its transport parameters.

After the Stateless Reset Token, the endpoint pads the message with an arbitrary number of octets containing random values.

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

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.

7.8.1. Detecting a Stateless Reset

A client detects a potential stateless reset when a packet with a short header cannot be decrypted. The client then performs a constant-time comparison of the 16 octets that follow the Connection ID with the Stateless Reset Token provided by the server in its transport parameters. If this comparison is successful, the connection MUST be terminated immediately. Otherwise, the packet can be discarded.

7.8.2. Calculating a Stateless Reset Token

The stateless reset token MUST be difficult to guess. In order to create a Stateless Reset Token, a server could randomly generate [RFC4086] a secret for every connection that it creates. However, this presents a coordination problem when there are multiple servers in a cluster or a storage problem for a server 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 three inputs: the static key, a the connection ID for the connection (see Section 5.6), and an identifier for the server instance. A server could use HMAC [RFC2104] (for example, HMAC(static_key, server_id || connection_id)) or HKDF [RFC5869] (for example, using the static key as input keying material, with server and connection identifiers as salt). The output of this function is truncated to 16 octets to produce the Stateless Reset Token for that connection.

A server that loses state can use the same method to generate a valid Stateless Reset Secret. The connection ID comes from the packet that the server receives.

This design relies on the client always sending a connection ID in its packets so that the server can use the connection ID from a packet to reset the connection. A server that uses this design

cannot allow clients to omit a connection ID (that is, it cannot use the truncate_connection_id transport parameter Section 7.3.1).

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 server instance, connection ID, and static key cannot occur for another connection. A connection ID from a connection that is reset by revealing the Stateless Reset Token cannot be reused for new connections at the same server without first changing to use a different static key or server identifier.

8. Frame Types and Formats

As described in Section 6, Regular packets contain one or more frames. We now describe the various QUIC frame types that can be present in a Regular packet. The use of these frames and various frame header bits are described in subsequent sections.

8.1. PADDING Frame

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

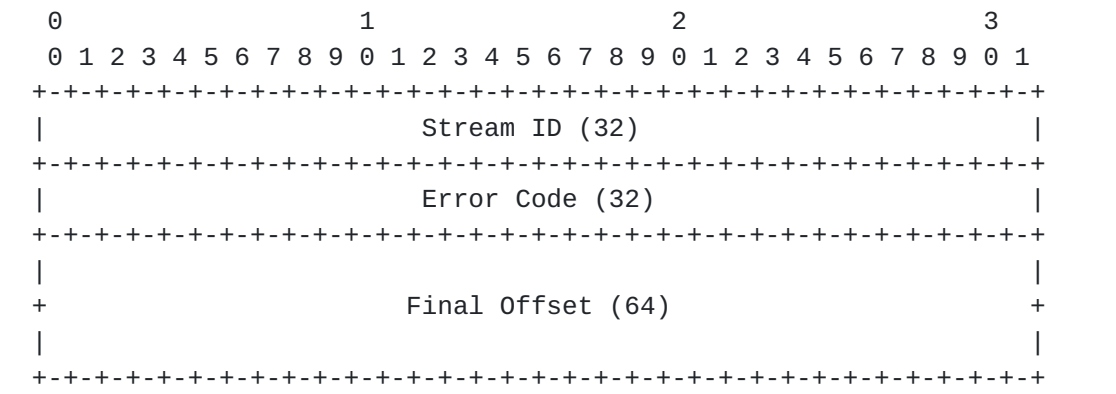
A PADDING frame has no content. That is, a PADDING frame consists of the single octet that identifies the frame as a PADDING frame.

8.2. RST_STREAM Frame

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

After sending a RST_STREAM, an endpoint ceases transmission of STREAM frames on the identified stream. A receiver of RST_STREAM can discard any data that it already received on that stream. An endpoint sends a RST_STREAM in response to a RST_STREAM unless the stream is already closed.

The RST_STREAM frame is as follows:



The fields are:

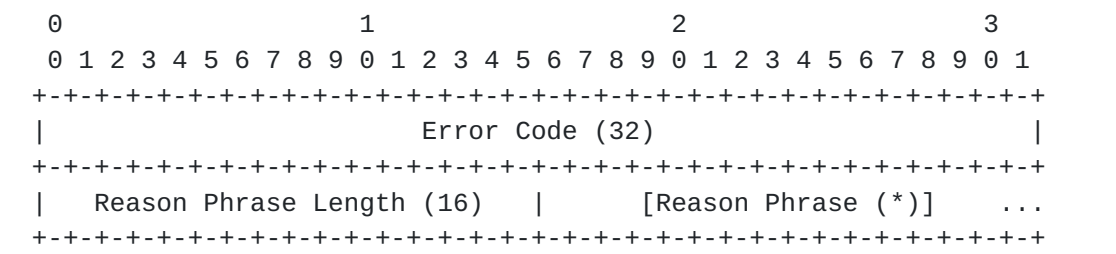
Stream ID: The 32-bit Stream ID of the stream being terminated.

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

Final offset: A 64-bit unsigned integer indicating the absolute byte offset of the end of data written on this stream by the RST_STREAM sender.

8.3. CONNECTION_CLOSE frame

An endpoint sends a CONNECTION_CLOSE frame (type=0x02) to notify its peer that the connection is being closed. If there are open streams that haven't been explicitly closed, they are implicitly closed when the connection is closed. The frame is as follows:



The fields of a CONNECTION_CLOSE frame are as follows:

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

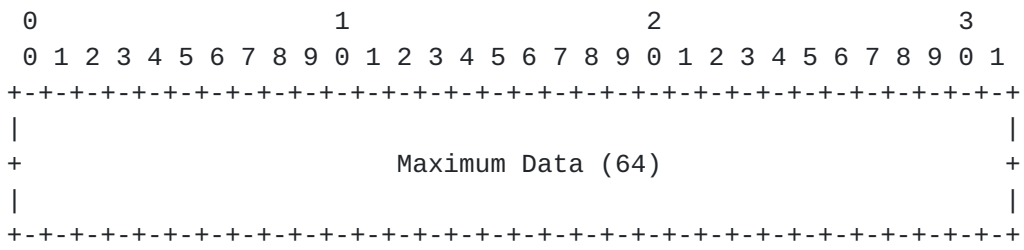
Reason Phrase Length: A 16-bit unsigned number specifying the length of the reason phrase in bytes. Note that a CONNECTION_CLOSE frame cannot be split between packets, so in practice any limits on packet size will also limit the space available for a reason phrase.

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

8.4. MAX_DATA Frame

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

The frame is as follows:



The fields in the MAX_DATA frame are as follows:

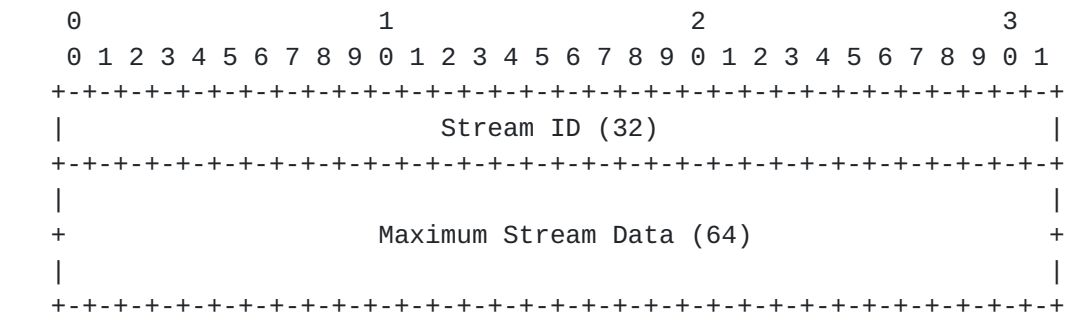
Maximum Data: A 64-bit unsigned integer indicating the maximum amount of data that can be sent on the entire connection, in units of 1024 octets. That is, the updated connection-level data limit is determined by multiplying the encoded value by 1024.

All data sent in STREAM frames counts toward this limit, with the exception of data on stream 0. The sum of the largest received offsets on all streams - including closed streams, but excluding stream 0 - MUST NOT exceed the value advertised by a receiver. An endpoint MUST terminate a connection with a QUIC_FLOW_CONTROL_RECEIVED_TOO_MUCH_DATA 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 7.3.2).

8.5. MAX_STREAM_DATA Frame

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

The frame is as follows:



The fields in the MAX_STREAM_DATA frame are as follows:

Stream ID: The stream ID of the stream that is affected.

Maximum Stream Data: A 64-bit unsigned integer indicating the maximum amount of data that can be sent on the identified stream, in units of octets.

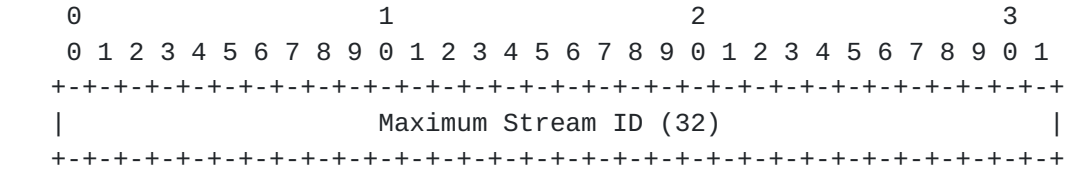
When counting data toward this limit, an endpoint accounts for the largest received offset of data that is sent or received on the stream. Loss or reordering can mean that the largest received offset on a stream can be greater than the total size of data received on that stream. Receiving STREAM frames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

The data sent on a stream MUST NOT exceed the largest maximum stream data value advertised by the receiver. An endpoint MUST terminate a connection with a FLOW_CONTROL_ERROR error if it receives more data than the largest maximum stream data that it has sent for the affected stream, unless this is a result of a change in the initial limits (see Section 7.3.2).

8.6. MAX_STREAM_ID Frame

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

The frame is as follows:



The fields in the MAX_STREAM_ID frame are as follows:

Maximum Stream ID: ID of the maximum peer-initiated stream ID for the connection.

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

A peer MUST NOT initiate a stream with a higher stream ID than the greatest maximum stream ID it has received. An endpoint MUST terminate a connection with a STREAM_ID_ERROR error if a peer initiates a stream with a higher stream ID than it has sent, unless this is a result of a change in the initial limits (see Section 7.3.2).

8.7. PING frame

Endpoints can use PING frames (type=0x07) to verify that their peers are still alive or to check reachability to the peer. The PING frame contains no additional fields. The receiver of a PING frame simply needs to acknowledge the packet containing this frame. The PING frame SHOULD be used to keep a connection alive when a stream is open. The default is to send a PING frame after 15 seconds of quiescence. A PING frame has no additional fields.

8.8. BLOCKED Frame

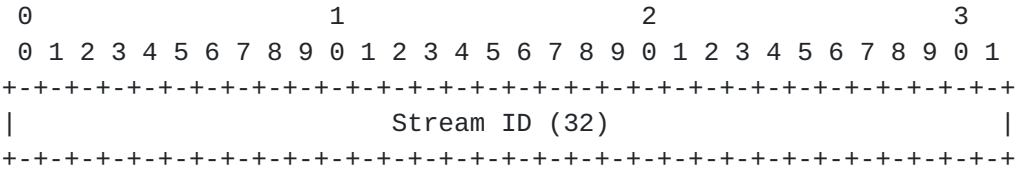
A sender sends a BLOCKED frame (type=0x08) when it wishes to send data, but is unable to due to connection-level flow control (see Section 11.2.1). BLOCKED frames can be used as input to tuning of flow control algorithms (see Section 11.1.2).

The BLOCKED frame does not contain a payload.

8.9. STREAM_BLOCKED Frame

A sender sends a STREAM_BLOCKED frame (type=0x09) when it wishes to send data, but is unable to due to stream-level flow control. This frame is analogous to BLOCKED (Section 8.8).

The STREAM_BLOCKED frame is as follows:



The STREAM_BLOCKED frame contains a single field:

Stream ID: A 32-bit unsigned number indicating the stream which is flow control blocked.

An endpoint MAY send a STREAM_BLOCKED frame for a stream that exceeds the maximum stream ID set by its peer (see Section 8.6). This does not open the stream, but informs the peer that a new stream was needed, but the stream limit prevented the creation of the stream.

8.10. STREAM_ID_NEEDED Frame

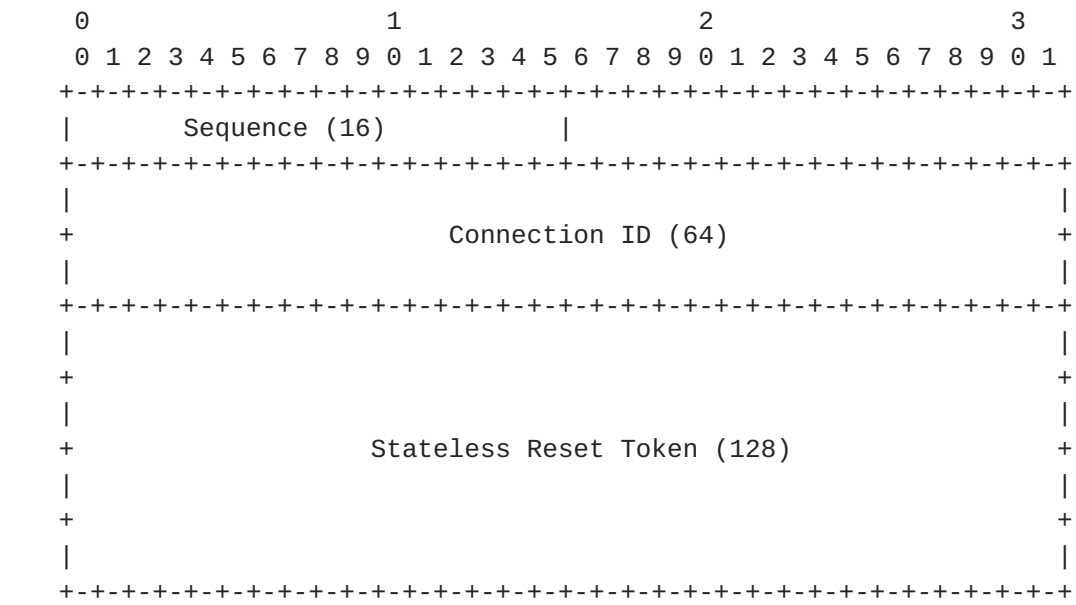
A sender sends a STREAM_ID_NEEDED frame (type=0x0a) when it wishes to open a stream, but is unable to due to the maximum stream ID limit.

The STREAM_ID_NEEDED frame does not contain a payload.

8.11. NEW_CONNECTION_ID Frame

A server sends a NEW_CONNECTION_ID frame (type=0x0b) to provide the client with alternative connection IDs that can be used to break linkability when migrating connections (see Section 7.6.1).

The NEW_CONNECTION_ID is as follows:



The fields are:

Sequence: A 16-bit sequence number. This value starts at 0 and increases by 1 for each connection ID that is provided by the server. The sequence value can wrap; the value 65535 is followed

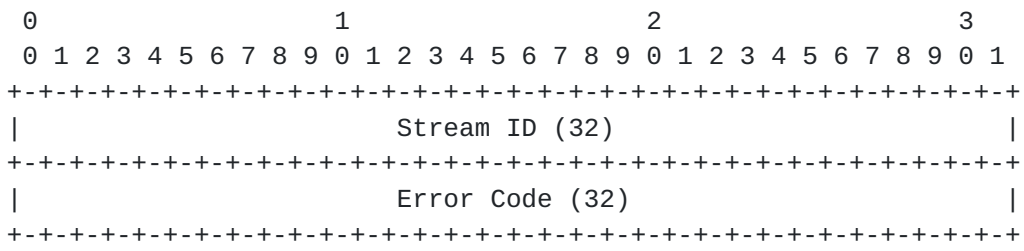
by 0. When wrapping the sequence field, the server MUST ensure that a value with the same sequence has been received and acknowledged by the client. The connection ID that is assigned during the handshake is assumed to have a sequence of 65535.

Connection ID: A 64-bit connection ID.

Stateless Reset Token: A 128-bit value that will be used to for a stateless reset when the associated connection ID is used (see Section 7.8).

8.12. STOP_SENDING Frame

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



The fields are:

Stream ID: The 32-bit Stream ID of the stream being ignored.

Error Code: The application-specified reason the sender is ignoring the stream.

8.13. ACK Frame

Receivers send ACK frames to inform senders which packets they have received and processed, as well as which packets are considered missing. The ACK frame contains between 1 and 256 ACK blocks. ACK blocks are ranges of acknowledged packets. Implementations SHOULD NOT generate ACK packets in response to packets which only contain ACKs. However, they SHOULD ACK those packets when sending ACKs for other packets.

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

A receiver that is only sending ACK frames will not receive acknowledgments for its packets. Sending an occasional MAX_DATA or MAX_STREAM_DATA frame as data is received will ensure that acknowledgements are generated by a peer. Otherwise, an endpoint MAY send a PING frame once per RTT to solicit an acknowledgment.

To limit receiver state or the size of ACK frames, a receiver MAY limit the number of ACK blocks it sends. A receiver can do this even without receiving acknowledgment of its ACK frames, with the knowledge this could cause the sender to unnecessarily retransmit some data. When this is necessary, the receiver SHOULD acknowledge newly received packets and stop acknowledging packets received in the past.

Unlike TCP SACKs, QUIC ACK blocks are irrevocable. Once a packet has been acknowledged, even if it does not appear in a future ACK frame, it remains acknowledged.

A client MUST NOT acknowledge Version Negotiation or Server Stateless Retry packets. These packet types contain packet numbers selected by the client, not the server.

QUIC ACK frames contain a timestamp section with up to 255 timestamps. Timestamps enable better congestion control, but are not required for correct loss recovery, and old timestamps are less valuable, so it is not guaranteed every timestamp will be received by the sender. A receiver SHOULD send a timestamp exactly once for each received packet containing retransmittable frames. A receiver MAY send timestamps for non-retransmittable packets. A receiver MUST not send timestamps in unprotected packets.

A sender MAY intentionally skip packet numbers to introduce entropy into the connection, to avoid opportunistic acknowledgement attacks. The sender SHOULD close the connection if an unsent packet number is acknowledged. The format of the ACK frame is efficient at expressing blocks of missing packets; skipping packet numbers between 1 and 255 effectively provides up to 8 bits of efficient entropy on demand, which should be adequate protection against most opportunistic acknowledgement attacks.

The type byte for a ACK frame contains embedded flags, and is formatted as "101NLLMM". These bits are parsed as follows:

- The first three bits must be set to 101 indicating that this is an ACK frame.

- The "N" bit indicates whether the frame contains a Num Blocks field.

- The two "LL" bits encode the length of the Largest Acknowledged field. The values 00, 01, 02, and 03 indicate lengths of 8, 16, 32, and 64 bits respectively.

- The two "MM" bits encode the length of the ACK Block Length fields. The values 00, 01, 02, and 03 indicate lengths of 8, 16, 32, and 64 bits respectively.

An ACK frame is shown below.

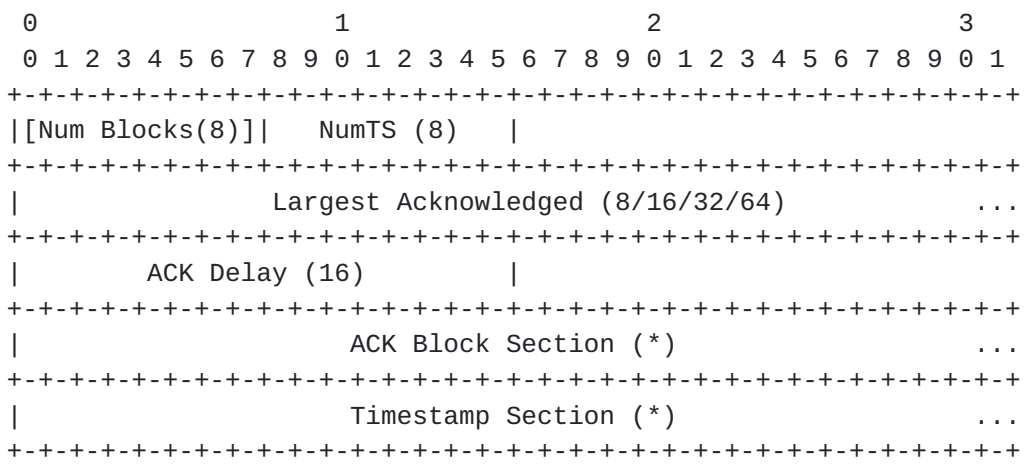


Figure 7: ACK Frame Format

The fields in the ACK frame are as follows:

Num Blocks (opt): An optional 8-bit unsigned value specifying the number of additional ACK blocks (besides the required First ACK Block) in this ACK frame. Only present if the 'N' flag bit is 1.

Num Timestamps: An unsigned 8-bit number specifying the total number of <packet number, timestamp> pairs in the Timestamp Section.

Largest Acknowledged: A variable-sized unsigned value representing the largest packet number the peer is acknowledging in this packet (typically the largest that the peer has seen thus far.)

ACK Delay: The time from when the largest acknowledged packet, as indicated in the Largest Acknowledged field, was received by this peer to when this ACK was sent.

ACK Block Section: Contains one or more blocks of packet numbers which have been successfully received, see Section 8.13.1.

Timestamp Section: Contains zero or more timestamps reporting transit delay of received packets. See Section 8.13.2.

8.13.1. ACK Block Section

The ACK Block Section contains between one and 256 blocks of packet numbers which have been successfully received. If the Num Blocks field is absent, only the First ACK Block length is present in this section. Otherwise, the Num Blocks field indicates how many additional blocks follow the First ACK Block Length field.

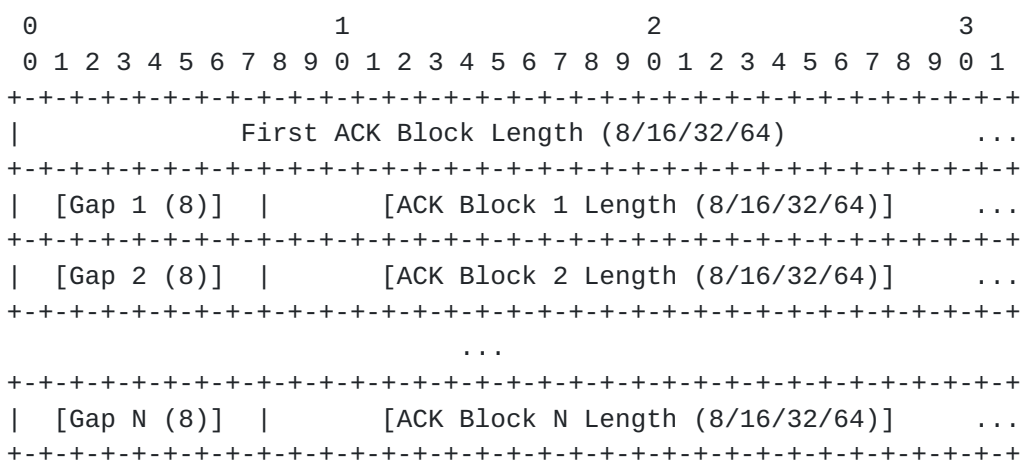


Figure 8: ACK Block Section

The fields in the ACK Block Section are:

First ACK Block Length: An unsigned packet number delta that indicates the number of contiguous additional packets being acknowledged starting at the Largest Acknowledged.

Gap To Next Block (opt, repeated): An unsigned number specifying the number of contiguous missing packets from the end of the previous ACK block to the start of the next. Repeated "Num Blocks" times.

ACK Block Length (opt, repeated): An unsigned packet number delta that indicates the number of contiguous packets being acknowledged starting after the end of the previous gap. Repeated "Num Blocks" times.

8.13.2. Timestamp Section

The Timestamp Section contains between zero and 255 measurements of packet receive times relative to the beginning of the connection.

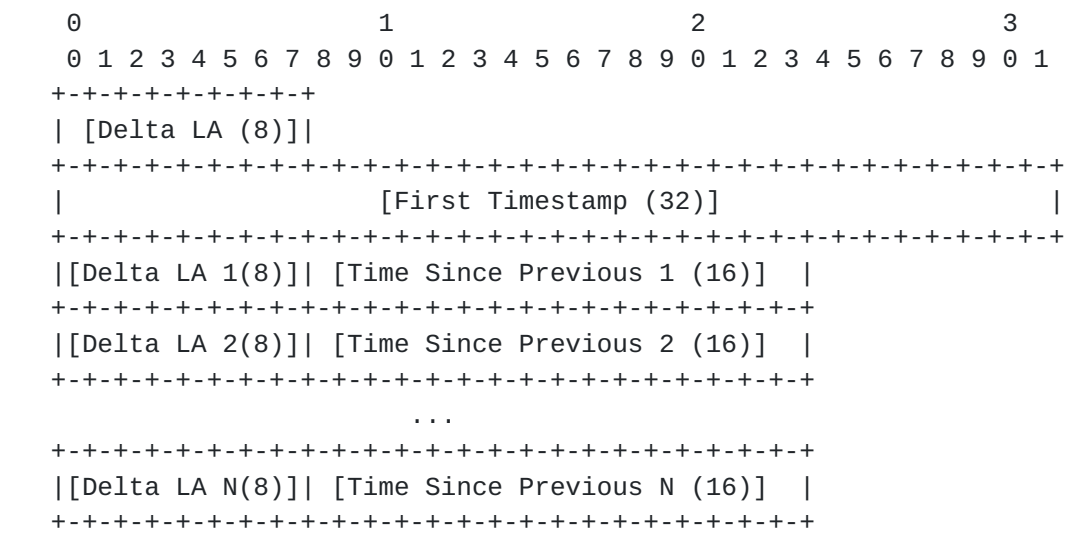


Figure 9: Timestamp Section

The fields in the Timestamp Section are:

Delta Largest Acknowledged (opt): An optional 8-bit unsigned packet number delta specifying the delta between the largest acknowledged and the first packet whose timestamp is being reported. In other words, this first packet number may be computed as (Largest Acknowledged - Delta Largest Acknowledged.)

First Timestamp (opt): An optional 32-bit unsigned value specifying the time delta in microseconds, from the beginning of the connection to the arrival of the packet indicated by Delta Largest Acknowledged.

Delta Largest Acked 1..N (opt, repeated): This field has the same semantics and format as "Delta Largest Acknowledged". Repeated "Num Timestamps - 1" times.

Time Since Previous Timestamp 1..N(opt, repeated): An optional 16-bit unsigned value specifying time delta from the previous reported timestamp. It is encoded in the same format as the ACK Delay. Repeated "Num Timestamps - 1" times.

The timestamp section lists packet receipt timestamps ordered by timestamp.

8.13.2.1. Time Format

DISCUSS_AND_REPLACE: Perhaps make this format simpler.

The time format used in the ACK frame above is a 16-bit unsigned float with 11 explicit bits of mantissa and 5 bits of explicit exponent, specifying time in microseconds. The bit format is loosely modeled after IEEE 754. For example, 1 microsecond is represented as 0x1, which has an exponent of zero, presented in the 5 high order bits, and mantissa of 1, presented in the 11 low order bits. When the explicit exponent is greater than zero, an implicit high-order 12th bit of 1 is assumed in the mantissa. For example, a floating value of 0x800 has an explicit exponent of 1, as well as an explicit mantissa of 0, but then has an effective mantissa of 4096 (12th bit is assumed to be 1). Additionally, the actual exponent is one-less than the explicit exponent, and the value represents 4096 microseconds. Any values larger than the representable range are clamped to 0xFFFF.

8.13.3. ACK Frames and Packet Protection

ACK frames that acknowledge protected packets MUST be carried in a packet that has an equivalent or greater level of packet protection.

Packets that are protected with 1-RTT keys MUST be acknowledged in packets that are also protected with 1-RTT keys.

A packet that is not protected and claims to acknowledge a packet number that was sent with packet protection is not valid. An unprotected packet that carries acknowledgments for protected packets MUST be discarded in its entirety.

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.

Unprotected packets, such as those that carry the initial cryptographic handshake messages, MAY be acknowledged in unprotected packets. Unprotected packets are vulnerable to falsification or modification. Unprotected packets can be acknowledged along with protected packets in a protected packet.

An endpoint SHOULD acknowledge packets containing cryptographic handshake messages in the next unprotected packet that it sends, unless it is able to acknowledge those packets in later packets protected by 1-RTT keys. At the completion of the cryptographic handshake, both peers send unprotected packets containing cryptographic handshake messages followed by packets protected by 1-RTT keys. An endpoint SHOULD acknowledge the unprotected packets

that complete the cryptographic handshake in a protected packet, because its peer is guaranteed to have access to 1-RTT packet protection keys.

For instance, a server acknowledges a TLS ClientHello in the packet that carries the TLS ServerHello; similarly, a client can acknowledge a TLS HelloRetryRequest in the packet containing a second TLS ClientHello. The complete set of server handshake messages (TLS ServerHello through to Finished) might be acknowledged by a client in protected packets, because it is certain that the server is able to decipher the packet.

8.14. STREAM Frame

STREAM frames implicitly create a stream and carry stream data. The type byte for a STREAM frame contains embedded flags, and is formatted as "11FSSOOD". These bits are parsed as follows:

- The first two bits must be set to 11, indicating that this is a STREAM frame.

- "F" is the FIN bit, which is used for stream termination.

- The "SS" bits encode the length of the Stream ID header field. The values 00, 01, 02, and 03 indicate lengths of 8, 16, 24, and 32 bits long respectively.

- The "OO" bits encode the length of the Offset header field. The values 00, 01, 02, and 03 indicate lengths of 0, 16, 32, and 64 bits long respectively.

- The "D" bit indicates whether a Data Length field is present in the STREAM header. When set to 0, this field indicates that the Stream Data field extends to the end of the packet. When set to 1, this field indicates that Data Length field contains the length (in bytes) of the Stream Data field. The option to omit the length should only be used when the packet is a "full-sized" packet, to avoid the risk of corruption via padding.

A STREAM frame is shown below.

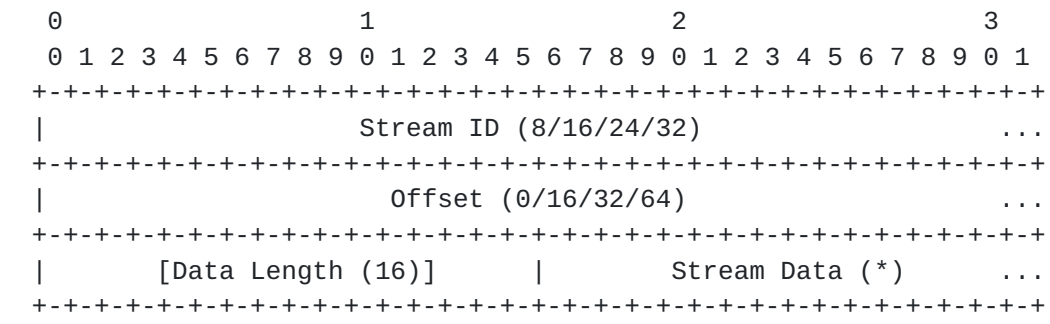


Figure 10: STREAM Frame Format

The STREAM frame contains the following fields:

Stream ID: The stream ID of the stream (see Section 10.1).

Offset: A variable-sized unsigned number specifying the byte offset in the stream for the data in this STREAM frame. When the offset length is 0, the offset is 0. The first byte in the stream has an offset of 0. The largest offset delivered on a stream - the sum of the re-constructed offset and data length - MUST be less than 2^64.

Data Length: An optional 16-bit unsigned number specifying the length of the Stream Data field in this STREAM frame. This field is present when the "D" bit is set to 1.

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

A stream frame's Stream Data MUST NOT be empty, unless the FIN bit is set. When the FIN flag is sent on an empty STREAM frame, the offset in the STREAM frame is the offset of the next byte that would be sent.

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.

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

9. Packetization and Reliability

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

All QUIC packets SHOULD be sized to fit within the estimated PMTU to avoid IP fragmentation or packet drops. To optimize bandwidth efficiency, endpoints SHOULD use Packetization Layer PMTU Discovery ([RFC4821]) and MAY use PMTU Discovery ([RFC1191], [RFC1981]) for detecting the PMTU, setting the PMTU appropriately, and storing the result of previous PMTU determinations.

In the absence of these mechanisms, QUIC endpoints SHOULD NOT send IP packets larger than 1280 octets. Assuming the minimum IP header size, this results in a QUIC packet size of 1232 octets for IPv6 and 1252 octets for IPv4.

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

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

Clients MUST ensure that the first packet in a connection, and any retransmissions of those octets, has a QUIC packet size of least 1200 octets. The packet size for a QUIC packet includes the QUIC header and integrity check, but not the UDP or IP header.

The initial client packet SHOULD be padded to exactly 1200 octets unless the client has a reasonable assurance that the PMTU is larger. Sending a packet of this size ensures that the network path supports an MTU of this size and helps reduce the amplitude of amplification attacks caused by server responses toward an unverified client address.

Servers MUST ignore an initial plaintext packet from a client if its total size is less than 1200 octets.

If a QUIC endpoint determines that the PMTU between any pair of local and remote IP addresses has fallen below 1280 octets, it MUST immediately cease sending QUIC packets on the affected path. This

could result in termination of the connection if an alternative path cannot be found.

A sender bundles one or more frames in a Regular QUIC packet (see Section 6).

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

Regular QUIC packets are "containers" of frames; a packet is never retransmitted whole. How an endpoint handles the loss of the frame depends on the type of the frame. Some frames are simply retransmitted, some have their contents moved to new frames, and others are never retransmitted.

When a packet is detected as lost, the sender re-sends any frames as necessary:

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

- ACK and PADDING frames MUST NOT be retransmitted. ACK frames containing updated information will be sent as described in Section 8.13.

- STOP_SENDING frames MUST be retransmitted, unless the stream has become closed in the appropriate direction. See Section 10.3.

- All other frames MUST be retransmitted.

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

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 queued

(but not necessarily delivered to the application). This also means that any stream state transitions triggered by STREAM or RST_STREAM frames have occurred. 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.

To avoid creating an indefinite feedback loop, an endpoint MUST NOT generate an ACK frame in response to a packet containing only ACK or PADDING frames.

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

9.1. Special Considerations for PMTU Discovery

Traditional ICMP-based path MTU discovery in IPv4 [RFC1191] is potentially vulnerable to off-path attacks that successfully guess the IP/port 4-tuple and reduce the MTU to a bandwidth-inefficient value. TCP connections mitigate this risk by using the (at minimum) 8 bytes of transport header echoed in the ICMP message to validate the TCP sequence number as valid for the current connection. However, as QUIC operates over UDP, in IPv4 the echoed information could consist only of the IP and UDP headers, which usually has insufficient entropy to mitigate off-path attacks.

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

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

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

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

10. Streams: QUIC's Data Structuring Abstraction

Streams in QUIC provide a lightweight, ordered, and bidirectional byte-stream abstraction modeled closely on HTTP/2 streams [RFC7540].

Streams can be created either by the client or the server, can concurrently send data interleaved with other streams, and can be cancelled.

Data that is received on a stream is delivered in order within that stream, but there is no particular delivery order across streams. Transmit ordering among streams is left to the implementation.

The creation and destruction of streams are expected to have minimal bandwidth and computational cost. A single STREAM frame may create, carry data for, and terminate a stream, or a stream may last the entire duration of a connection.

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

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

10.1. Stream Identifiers

Streams are identified by an unsigned 32-bit integer, referred to as the Stream ID. To avoid Stream ID collision, clients initiate streams using odd-numbered Stream IDs; streams initiated by the server use even-numbered Stream IDs.

Stream ID 0 (0x0) is reserved for the cryptographic handshake. Stream 0 MUST NOT be used for application data, and is the first client-initiated stream.

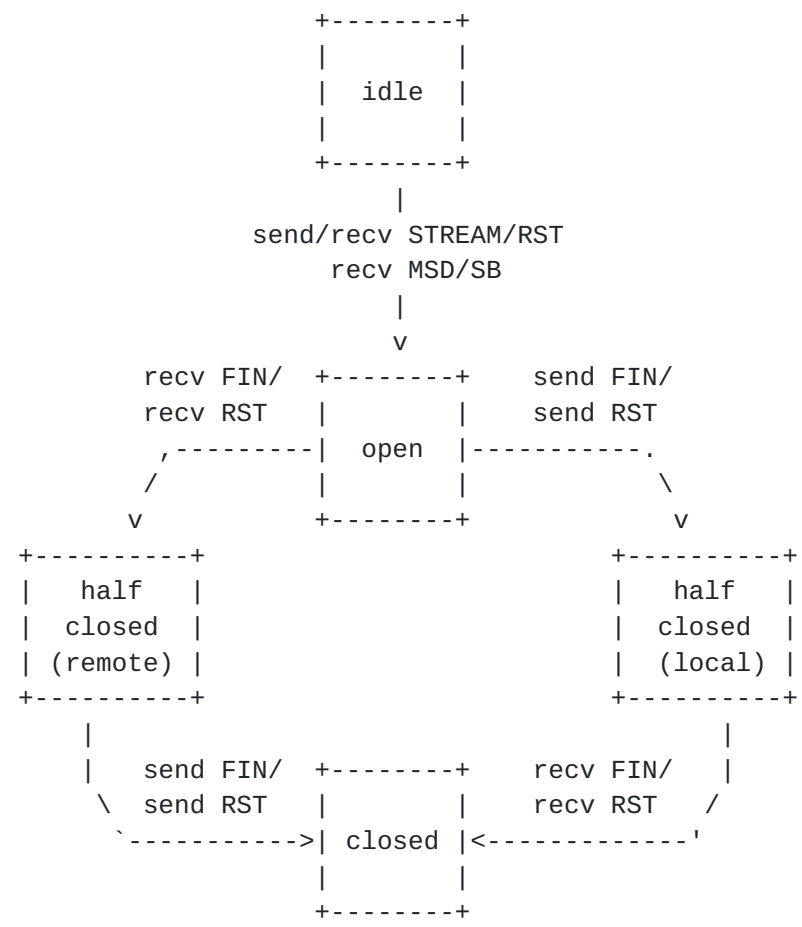
A QUIC endpoint cannot reuse a Stream ID. Streams MUST be created in sequential order. Open streams can be used in any order. Streams that are used out of order result in lower-numbered streams in the same direction being counted as open.

Stream IDs are usually encoded as a 32-bit integer, though the STREAM frame (Section 8.14) permits a shorter encoding when the leading bits of the stream ID are zero.

10.2. Life of a Stream

The semantics of QUIC streams is based on HTTP/2 streams, and the lifecycle of a QUIC stream therefore closely follows that of an HTTP/2 stream [RFC7540], with some differences to accommodate the

possibility of out-of-order delivery due to the use of multiple streams in QUIC. The lifecycle of a QUIC stream is shown in the following figure and described below.



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

STREAM: a STREAM frame
FIN: FIN flag in a STREAM frame
RST: RST_STREAM frame
MSD: MAX_STREAM_DATA frame
SB: STREAM_BLOCKED frame

Figure 11: Lifecycle of a stream

Note that this diagram shows stream state transitions and the frames and flags that affect those transitions only. It is possible for a single frame to cause two transitions: receiving a RST_STREAM frame, or a STREAM frame with the FIN flag cause the stream state to move from "idle" to "open" and then immediately to one of the "half-closed" states.

The recipient of a frame that changes stream state will have a delayed view of the state of a stream while the frame is in transit. Endpoints do not coordinate the creation of streams; they are created unilaterally by either endpoint. Endpoints can use acknowledgments to understand the peer's subjective view of stream state at any given time.

In the absence of more specific guidance elsewhere in this document, implementations SHOULD treat the receipt of a frame that is not expressly permitted in the description of a state as a connection error (see Section 12).

10.2.1. idle

All streams start in the "idle" state.

The following transitions are valid from this state:

Sending or receiving a STREAM or RST_STREAM frame causes the identified stream to become "open". The stream identifier for a new stream is selected as described in Section 10.1. A RST_STREAM frame, or a STREAM frame with the FIN flag set also causes a stream to become "half-closed".

An endpoint might receive MAX_STREAM_DATA or STREAM_BLOCKED frames on peer-initiated streams that are "idle" if there is loss or reordering of packets. Receiving these frames also causes the stream to become "open".

An endpoint MUST NOT send a STREAM or RST_STREAM frame for a stream ID that is higher than the peers advertised maximum stream ID (see Section 8.6).

10.2.2. open

A stream in the "open" state may be used by both peers to send frames of any type. In this state, endpoints can send MAX_STREAM_DATA and MUST observe the value advertised by its receiving peer (see Section 11).

Opening a stream causes all lower-numbered streams in the same direction to become open. Thus, opening an odd-numbered stream causes all "idle", odd-numbered streams with a lower identifier to become open and the same applies to even numbered streams. Endpoints open streams in increasing numeric order, but loss or reordering can cause packets that open streams to arrive out of order.

From the "open" state, either endpoint can send a frame with the FIN flag set, which causes the stream to transition into one of the "half-closed" states. This flag can be set on the frame that opens the stream, which causes the stream to immediately become "half-closed". Once an endpoint has completed sending all stream data and a STREAM frame with a FIN flag, the stream state becomes "half-closed (local)". When an endpoint receives all stream data and a FIN flag the stream state becomes "half-closed (remote)". An endpoint MUST NOT consider the stream state to have changed until all data has been sent or received.

A RST_STREAM frame on an "open" stream also causes the stream to become "half-closed". A stream that becomes "open" as a result of sending or receiving RST_STREAM immediately becomes "half-closed". Sending a RST_STREAM frame causes the stream to become "half-closed (local)"; receiving RST_STREAM causes the stream to become "half-closed (remote)".

Any frame type that mentions a stream ID can be sent in this state.

10.2.3. half-closed (local)

A stream that is in the "half-closed (local)" state MUST NOT be used for sending on new STREAM frames. Retransmission of data that has already been sent on STREAM frames is permitted. An endpoint MAY also send MAX_STREAM_DATA and STOP_SENDING in this state.

An endpoint that closes a stream MUST NOT send data beyond the final offset that it has chosen, see Section 10.2.5 for details.

A stream transitions from this state to "closed" when a STREAM frame that contains a FIN flag is received and all prior data has arrived, or when a RST_STREAM frame is received.

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

10.2.4. half-closed (remote)

A stream is "half-closed (remote)" when the stream is no longer being used by the peer to send any data. An endpoint will have either received all data that a peer has sent or will have received a RST_STREAM frame and discarded any received data.

Once all data has been either received or discarded, a sender is no longer obligated to update the maximum received data for the connection.

Due to reordering, an endpoint could continue receiving frames for the stream even after the stream is closed for sending. Frames received after a peer closes a stream SHOULD be discarded. An endpoint MAY choose to limit the period over which it ignores frames and treat frames that arrive after this time as being in error.

An endpoint will know the final offset of the data it receives on a stream when it reaches the "half-closed (remote)" state, see Section 11.3 for details.

A stream in this state can be used by the endpoint to send any frame that mentions a stream ID. In this state, the endpoint MUST observe advertised stream and connection data limits (see Section 11).

A stream transitions from this state to "closed" by completing transmission of all data. This includes sending all data carried in STREAM frames including the terminal STREAM frame that contains a FIN flag.

A stream also becomes "closed" when the endpoint sends a RST_STREAM frame.

10.2.5. closed

The "closed" state is the terminal state for a stream.

Once a stream reaches this state, no frames can be sent that mention the stream. Reordering might cause frames to be received after closing, see Section 10.2.4.

10.3. Solicited State Transitions

If an endpoint is no longer interested in the data being received, it MAY send a STOP_SENDING frame on a stream in the "open" or "half-closed (local)" state to prompt closure of the stream in the opposite direction. This typically indicates that the receiving application is no longer reading from the stream, but is not a guarantee that incoming data will be ignored.

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

Upon receipt of a STOP_SENDING frame on a stream in the "open" or "half-closed (remote)" states, an endpoint MUST send a RST_STREAM with an error code of QUIC_RECEIVED_RST. If the STOP_SENDING frame is received on a stream that is already in the "half-closed (local)" or "closed" states, a RST_STREAM frame MAY still be sent in order to cancel retransmission of previously-sent STREAM frames.

While STOP_SENDING frames are retransmittable, an implementation MAY choose not to retransmit a lost STOP_SENDING frame if the stream has already been closed in the appropriate direction since the frame was first generated. See Section 9.

10.4. Stream Concurrency

An endpoint limits the number of concurrently active incoming streams by adjusting the maximum stream ID. An initial value is set in the transport parameters (see Section 7.3.1) and is subsequently increased by MAX_STREAM_ID frames (see Section 8.6).

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

Endpoints MUST NOT exceed the limit set by their peer. An endpoint that receives a STREAM frame with an ID greater than the limit it has sent MUST treat this as a stream error of type STREAM_ID_ERROR (Section 12), unless this is a result of a change in the initial offsets (see Section 7.3.2).

A receiver MUST NOT renege on an advertisement; that is, once a receiver advertises a stream ID via a MAX_STREAM_ID frame, it MUST NOT subsequently advertise a smaller maximum ID. A sender may receive MAX_STREAM_ID frames out of order; a sender MUST therefore ignore any MAX_STREAM_ID that does not increase the maximum.

10.5. Sending and Receiving Data

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

When new data is to be sent on a stream, a sender MUST set the encapsulating STREAM frame's offset field to the stream offset of the first byte of this new data. The first byte of data that is sent on a stream has the stream offset 0. The largest offset delivered on a stream MUST be less than 2^64. A receiver MUST ensure that received stream data is delivered to the application as an ordered byte-stream. Data received out of order MUST be buffered for later delivery, as long as it is not in violation of the receiver's flow control limits.

An endpoint MUST NOT send data on any stream without ensuring that it is within the data limits set by its peer. The cryptographic handshake stream, Stream 0, is exempt from the connection-level data limits established by MAX_DATA. Stream 0 is still subject to stream-level data limits and MAX_STREAM_DATA.

Flow control is described in detail in Section 11, and congestion control is described in the companion document [QUIC-RECOVERY].

10.6. Stream Prioritization

Stream multiplexing has a significant effect on application performance if resources allocated to streams are correctly prioritized. Experience with other multiplexed protocols, such as HTTP/2 [RFC7540], shows that effective prioritization strategies have a significant positive impact on performance.

QUIC does not provide frames for exchanging prioritization information. Instead it relies on receiving priority information from the application that uses QUIC. Protocols that use QUIC are able to define any prioritization scheme that suits their application semantics. A protocol might define explicit messages for signaling priority, such as those defined in HTTP/2; it could define rules that allow an endpoint to determine priority based on context; or it could leave the determination to the application.

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

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

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

Stream 0 MUST be prioritized over other streams prior to the completion of the cryptographic handshake. This includes the retransmission of the second flight of client handshake messages, that is, the TLS Finished and any client authentication messages.

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

11. Flow Control

It is necessary to limit the amount of data that a sender may have outstanding at any time, so as to prevent a fast sender from overwhelming a slow receiver, or to prevent a malicious sender from consuming significant resources at a receiver. This section describes QUIC's flow-control mechanisms.

QUIC employs a credit-based flow-control scheme similar to HTTP/2's flow control [RFC7540]. A receiver advertises the number of octets it is prepared to receive on a given stream and for the entire connection. This leads to two levels of flow control in QUIC: (i) Connection flow control, which prevents senders from exceeding a receiver's buffer capacity for the connection, and (ii) Stream flow control, which prevents a single stream from consuming the entire receive buffer for a connection.

A receiver sends MAX_DATA or MAX_STREAM_DATA frames to the sender to advertise additional credit by sending the absolute byte offset in the connection or stream which it is willing to receive.

A receiver MAY advertise a larger offset at any point by sending MAX_DATA or MAX_STREAM_DATA frames. A receiver MUST NOT renege on an advertisement; that is, once a receiver advertises an offset, it MUST NOT subsequently advertise a smaller offset. A sender could receive MAX_DATA or MAX_STREAM_DATA frames out of order; a sender MUST therefore ignore any flow control offset that does not move the window forward.

A receiver MUST close the connection with a FLOW_CONTROL_ERROR error (Section 12) if the peer violates the advertised connection or stream data limits.

A sender MUST send BLOCKED frames to indicate it has data to write but is blocked by lack of connection or stream flow control credit. BLOCKED frames are expected to be sent infrequently in common cases, but they are considered useful for debugging and monitoring purposes.

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

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

11.1. Edge Cases and Other Considerations

There are some edge cases which must be considered when dealing with stream and connection level flow control. Given enough time, both endpoints must agree on flow control state. If one end believes it can send more than the other end is willing to receive, the connection will be torn down when too much data arrives.

Conversely if a sender believes it is blocked, while endpoint B expects more data can be received, then the connection can be in a deadlock, with the sender waiting for a MAX_DATA or MAX_STREAM_DATA frame which will never come.

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

To avoid this de-synchronization, a RST_STREAM sender MUST include the final byte offset sent on the stream in the RST_STREAM frame. On receiving a RST_STREAM frame, a receiver definitively knows how many

bytes were sent on that stream before the RST_STREAM frame, and the receiver MUST use the final offset to account for all bytes sent on the stream in its connection level flow controller.

11.1.1. Response to a RST_STREAM

RST_STREAM terminates one direction of a stream abruptly. Whether any action or response can or should be taken on the data already received is an application-specific issue, but it will often be the case that upon receipt of a RST_STREAM an endpoint will choose to stop sending data in its own direction. If the sender of a RST_STREAM wishes to explicitly state that no future data will be processed, that endpoint MAY send a STOP_SENDING frame at the same time.

11.1.2. Data Limit Increments

This document leaves when and how many bytes to advertise in a MAX_DATA or MAX_STREAM_DATA to implementations, but offers a few considerations. These frames contribute to connection overhead. Therefore frequently sending frames with small changes is undesirable. At the same time, infrequent updates require larger increments to limits if blocking is to be avoided. Thus, larger updates require a receiver to commit to larger resource commitments. Thus there is a tradeoff between resource commitment and overhead when determining how large a limit is advertised.

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

11.2. Stream Limit Increment

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

Implementations will likely want to increase the maximum stream ID as peer-initiated streams close. A receiver MAY also advance the maximum stream ID based on current activity, system conditions, and other environmental factors.

11.2.1. Blocking on Flow Control

If a sender does not receive a MAX_DATA or MAX_STREAM_DATA frame when it has run out of flow control credit, the sender will be blocked and MUST send a BLOCKED or STREAM_BLOCKED frame. These frames are expected to be useful for debugging at the receiver; they do not require any other action. A receiver SHOULD NOT wait for a BLOCKED or STREAM_BLOCKED frame before sending MAX_DATA or MAX_STREAM_DATA, since doing so will mean that a sender is unable to send for an entire round trip.

For smooth operation of the congestion controller, it is generally considered best to not let the sender go into quiescence if avoidable. To avoid blocking a sender, and to reasonably account for the possibiity of loss, a receiver should send a MAX_DATA or MAX_STREAM_DATA frame at least two roundtrips before it expects the sender to get blocked.

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

11.3. Stream Final Offset

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

An endpoint will know the final offset for a stream when the stream enters the "half-closed (remote)" state. However, if there is reordering or loss, an endpoint might learn the final offset prior to entering this state if it is carried on a STREAM frame.

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

Once a final offset for a stream is known, it cannot change. If a RST_STREAM or STREAM frame causes the final offset to change for a stream, an endpoint SHOULD respond with a FINAL_OFFSET_ERROR error (see Section 12). A receiver SHOULD treat receipt of data at or beyond the final offset as a FINAL_OFFSET_ERROR error, even after a stream is closed. Generating these errors is not mandatory, but only because requiring that an endpoint generate these errors also means

that the endpoint needs to maintain the final offset state for closed streams, which could mean a significant state commitment.

12. Error Handling

An endpoint that detects an error SHOULD signal the existence of that error to its peer. Errors can affect an entire connection (see Section 12.1), or a single stream (see Section 12.2).

The most appropriate error code (Section 12.3) SHOULD be included in the frame that signals the error. Where this specification identifies error conditions, it also identifies the error code that is used.

A stateless reset (Section 7.8) is not suitable for any error that can be signaled with a CONNECTION_CLOSE or RST_STREAM frame. A stateless reset MUST NOT be used by an endpoint that has the state necessary to send a frame on the connection.

12.1. 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 8.3). An endpoint MAY close the connection in this manner, even if the error only affects a single stream.

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 CONNECTION_CLOSE risks a peer missing the first such packet. The only mechanism available to an endpoint that continues to receive data for a terminated connection is to use the stateless reset process (Section 7.8).

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

12.2. Stream Errors

If the error affects a single stream, but otherwise leaves the connection in a recoverable state, the endpoint can send a RST_STREAM frame (Section 8.2) with an appropriate error code to terminate just the affected stream.

Stream 0 is critical to the functioning of the entire connection. If stream 0 is closed with either a RST_STREAM or STREAM frame bearing the FIN flag, an endpoint MUST generate a connection error of type PROTOCOL_VIOLATION.

Some application protocols make other streams critical to that protocol. An application protocol does not need to inform the transport that a stream is critical; it can instead generate appropriate errors in response to being notified that the critical stream is closed.

An endpoint MAY send a RST_STREAM frame in the same packet as a CONNECTION_CLOSE frame.

12.3. Error Codes

Error codes are 32 bits long, with the first two bits indicating the source of the error code:

0x00000000-0x3FFFFFFF: Application-specific error codes. Defined by each application-layer protocol.

0x40000000-0x7FFFFFFF: Reserved for host-local error codes. These codes MUST NOT be sent to a peer, but MAY be used in API return codes and logs.

0x80000000-0xBFFFFFFF: QUIC transport error codes, including packet protection errors. Applicable to all uses of QUIC.

0xC0000000-0xFFFFFFFF: Cryptographic error codes. Defined by the cryptographic handshake protocol in use.

This section lists the defined QUIC transport error codes that may be used in a CONNECTION_CLOSE or RST_STREAM frame. Error codes share a common code space. Some error codes apply only to either streams or the entire connection and have no defined semantics in the other context.

NO_ERROR (0x80000000): An endpoint uses this with CONNECTION_CLOSE to signal that the connection is being closed abruptly in the absence of any error. An endpoint uses this with RST_STREAM to

signal that the stream is no longer wanted or in response to the receipt of a RST_STREAM for that stream.

INTERNAL_ERROR (0x80000001): The endpoint encountered an internal error and cannot continue with the connection.

CANCELLED (0x80000002): An endpoint sends this with RST_STREAM to indicate that the stream is not wanted and that no application action was taken for the stream. This error code is not valid for use with CONNECTION_CLOSE.

FLOW_CONTROL_ERROR (0x80000003): An endpoint received more data than it permitted in its advertised data limits (see Section 11).

STREAM_ID_ERROR (0x80000004): An endpoint received a frame for a stream identifier that exceeded its advertised maximum stream ID.

STREAM_STATE_ERROR (0x80000005): An endpoint received a frame for a stream that was not in a state that permitted that frame (see Section 10.2).

FINAL_OFFSET_ERROR (0x80000006): An endpoint received a STREAM frame containing data that exceeded the previously established final offset. Or an endpoint received a RST_STREAM frame containing a final offset that was lower than the maximum offset of data that was already received. Or an endpoint received a RST_STREAM frame containing a different final offset to the one already established.

FRAME_FORMAT_ERROR (0x80000007): An endpoint received a frame that was badly formatted. For instance, an empty STREAM frame that omitted the FIN flag, or an ACK frame that has more acknowledgment ranges than the remainder of the packet could carry. This is a generic error code; an endpoint SHOULD use the more specific frame format error codes (0x800001XX) if possible.

TRANSPORT_PARAMETER_ERROR (0x80000008): An endpoint received transport parameters that were badly formatted, included an invalid value, was absent even though it is mandatory, was present though it is forbidden, or is otherwise in error.

VERSION_NEGOTIATION_ERROR (0x80000009): An endpoint received transport parameters that contained version negotiation parameters that disagreed with the version negotiation that it performed. This error code indicates a potential version downgrade attack.

PROTOCOL_VIOLATION (0x8000000A): An endpoint detected an error with protocol compliance that was not covered by more specific error codes.

QUIC_RECEIVED_RST (0x80000035): Terminating stream because peer sent a RST_STREAM or STOP_SENDING.

FRAME_ERROR (0x800001XX): An endpoint detected an error in a specific frame type. The frame type is included as the last octet of the error code. For example, an error in a MAX_STREAM_ID frame would be indicated with the code (0x80000106).

13. Security and Privacy Considerations

13.1. Spoofed ACK Attack

An attacker receives an STK from the server and then releases the IP address on which it received the STK. The attacker may, in the future, spoof this same address (which now presumably addresses a different endpoint), and initiate a 0-RTT connection with a server on the victim's behalf. The attacker then spoofs ACK frames to the server which cause the server to potentially drown the victim in data.

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

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

13.2. 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.

13.3. Stream Fragmentation and Reassembly Attacks

An adversarial endpoint might intentionally fragment the data on stream buffers in order to cause disproportionate memory commitment. An adversarial endpoint could open a stream and send some STREAM frames containing arbitrary fragments of the stream content.

The attack 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 the stream fragmentation attack. Mitigations could consist of avoiding over-committing memory, delaying reassembly of STREAM frames, implementing heuristics based on the age and duration of reassembly holes, or some combination.

13.4. 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 10.1. However, when several streams are initiated at short intervals, transmission error may cause STREAM DATA frames opening streams to be received out of sequence. A receiver is obligated to open intervening streams if a higher-numbered stream ID is received. Thus, on a new connection, opening stream 2000001 opens 1 million streams, as required by the specification.

The number of active streams is limited by the concurrent stream limit transport parameter, as explained in Section 10.4. If chosen judisciously, this limit mitigates the effect of the stream commitment attack. However, setting the limit too low could affect performance when applications expect to open large number of streams.

14. IANA Considerations

14.1. 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 [RFC5226]. Values with the first byte 0xff are reserved for Private Use [RFC5226].

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. The 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 4.

Value	Parameter Name	Specification
0x0000	initial_max_stream_data	Section 7.3.1
0x0001	initial_max_data	Section 7.3.1
0x0002	initial_max_stream_id	Section 7.3.1
0x0003	idle_timeout	Section 7.3.1
0x0004	omit_connection_id	Section 7.3.1
0x0005	max_packet_size	Section 7.3.1
0x0006	stateless_reset_token	Section 7.3.1

Table 4: Initial QUIC Transport Parameters Entries

15. References

15.1. Normative References

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-21 (work in progress), July 2017.

[QUIC-RECOVERY]
Iyengar, J., Ed. and I. Swett, Ed., "QUIC Loss Detection and Congestion Control", draft-ietf-quic-recovery (work in progress), August 2017.

[QUIC-TLS]
Thomson, M., Ed. and S. Turner, Ed., "Using Transport Layer Security (TLS) to Secure QUIC", draft-ietf-quic-tls (work in progress), August 2017.

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

[RFC1981] McCann, J., Deering, S., and J. Mogul, "Path MTU Discovery for IP version 6", RFC 1981, DOI 10.17487/RFC1981, August 1996, <http://www.rfc-editor.org/info/rfc1981>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <http://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, <http://www.rfc-editor.org/info/rfc4086>.

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

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", RFC 5226, DOI 10.17487/RFC5226, May 2008, <http://www.rfc-editor.org/info/rfc5226>.

15.2. Informative References

[EARLY-DESIGN] Roskind, J., "QUIC: Multiplexed Transport Over UDP", December 2013, <https://goo.gl/dMVtFi>.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <http://www.rfc-editor.org/info/rfc2104>.

[RFC2360] Scott, G., "Guide for Internet Standards Writers", BCP 22, RFC 2360, DOI 10.17487/RFC2360, June 1998, <http://www.rfc-editor.org/info/rfc2360>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <http://www.rfc-editor.org/info/rfc5869>.

[RFC6824] Ford, A., Raiciu, C., Handley, M., and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses", RFC 6824, DOI 10.17487/RFC6824, January 2013, <http://www.rfc-editor.org/info/rfc6824>.

[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, <http://www.rfc-editor.org/info/rfc7301>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <http://www.rfc-editor.org/info/rfc7540>.

[SLOWLORIS] RSnake Hansen, R., "Welcome to Slowloris...", June 2009, <https://web.archive.org/web/20150315054838/http://ha.ckers.org/slowloris/>.

[SST] Ford, B., "Structured streams", ACM SIGCOMM Computer Communication Review Vol. 37, pp. 361, DOI 10.1145/1282427.1282421, October 2007.

15.3. URIs

[1] https://github.com/quicwg/base-drafts/wiki/QUIC-Versions

Appendix A. 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.

Appendix B. 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.

Appendix C. 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.

C.1. Since draft-ietf-quic-transport-04

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

C.2. Since draft-ietf-quic-transport-03

- Change STREAM and RST_STREAM layout
- Add MAX_STREAM_ID settings

C.3. Since draft-ietf-quic-transport-02

- The size of the initial packet payload has a fixed minimum (#267, #472)
- Define when Version Negotiation packets are ignored (#284, #294, #241, #143, #474)
- The 64-bit FNV-1a algorithm is used for integrity protection of unprotected packets (#167, #480, #481, #517)
- Rework initial packet types to change how the connection ID is chosen (#482, #442, #493)
- No timestamps are forbidden in unprotected packets (#542, #429)
- Cryptographic handshake is now on stream 0 (#456)
- Remove congestion control exemption for cryptographic handshake (#248, #476)
- Version 1 of QUIC uses TLS; a new version is needed to use a different handshake protocol (#516)
- STREAM frames have a reduced number of offset lengths (#543, #430)
- Split some frames into separate connection- and stream- level frames (#443)
  - WINDOW_UPDATE split into MAX_DATA and MAX_STREAM_DATA (#450)
  - BLOCKED split to match WINDOW_UPDATE split (#454)
  - Define STREAM_ID_NEEDED frame (#455)
- A NEW_CONNECTION_ID frame supports connection migration without linkability (#232, #491, #496)
- Transport parameters for 0-RTT are retained from a previous connection (#405, #513, #512)
  - A client in 0-RTT no longer required to reset excess streams (#425, #479)
- Expanded security considerations (#440, #444, #445, #448)

C.4. Since draft-ietf-quic-transport-01

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

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

C.5. Since draft-ietf-quic-transport-00

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

C.6. Since draft-hamilton-quic-transport-protocol-01

- Adopted as base for draft-ietf-quic-tls
- Updated authors/editors list
- Added IANA Considerations section
- Moved Contributors and Acknowledgments to appendices

Authors' Addresses

Jana Iyengar (editor)
Google

Email: jri@google.com

Martin Thomson (editor)
Mozilla

Email: martin.thomson@gmail.com