QUIC
Internet-Draft
Intended status: Standards Track
Expires: October 19, 2018

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

April 17, 2018

# QUIC: A UDP-Based Multiplexed and Secure Transport draft-ietf-quic-transport-11

## 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 [1].

Working Group information can be found at https://github.com/quicwg [2]; source code and issues list for this draft can be found at https://github.com/quicwg/base-drafts/labels/-transport [3].

## Status of This Memo

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

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

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

This Internet-Draft will expire on October 19, 2018.

Table of Contents

## 1. Introduction

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

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

QUIC 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, connection migration, 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].

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

## 2. Conventions and Definitions

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

Definitions of terms that are used in this document:

Client: The endpoint initiating a QUIC connection.

Server: The endpoint accepting incoming QUIC connections.

Endpoint: The client or server end of a connection.

Stream: A logical, 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: An opaque identifier that is used to identify a QUIC connection at an endpoint. Each endpoint sets a value that its peer includes in packets.

QUIC packet: A well-formed UDP payload that can be parsed by a QUIC receiver.

QUIC is a name, not an acronym.

### 2.1. Notational Conventions

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

[x] Indicates that x is optional

x (A) Indicates that x is A bits long

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

x (i) ... Indicates that x uses the variable-length encoding in Section 7.1

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

## 3. Versions

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

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

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

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

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

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

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

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

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

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

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

## 4. 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 are used after version negotiation and 1-RTT keys are established.

### 4.1. Long Header

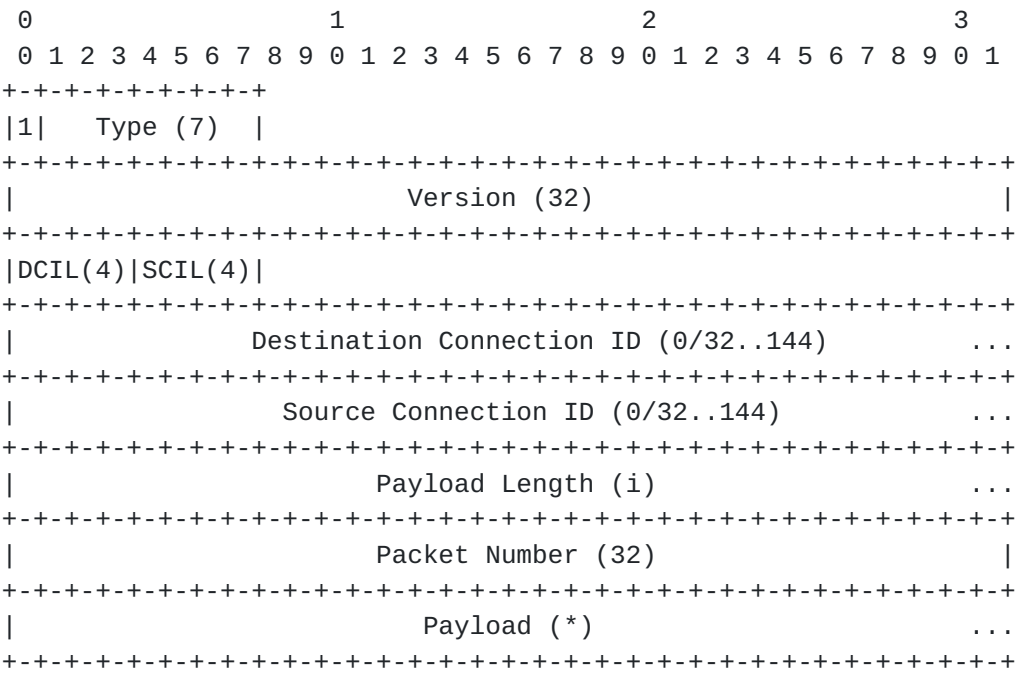



Figure 1: Long Header Format

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 switches to sending packets using the short header (Section 4.2). 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.

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

DCIL and SCIL: Octet 1 contains the lengths of the two connection ID fields that follow it. These lengths are encoded as two 4-bit unsigned integers. The Destination Connection ID Length (DCIL) field occupies the 4 high bits of the octet and the Source Connection ID Length (SCIL) field occupies the 4 low bits of the octet. An encoded length of 0 indicates that the connection ID is also 0 octets in length. Non-zero encoded lengths are increased by 3 to get the full length of the connection ID, producing a length between 4 and 18 octets inclusive. For example, an octet with the value 0x50 describes an 8-octet Destination Connection ID and a zero-length Source Connection ID.

Destination Connection ID: The Destination Connection ID field follows the connection ID lengths and is either 0 octets in length or between 4 and 18 octets. Section 4.7 describes the use of this field in more detail.

Source Connection ID: The Source Connection ID field follows the Destination Connection ID and is either 0 octets in length or between 4 and 18 octets. Section 4.7 describes the use of this field in more detail.

Payload Length: The length of the Payload field in octets, encoded as a variable-length integer (Section 7.1).

Packet Number: The Packet Number is a 32-bit field that follows the two connection IDs. Section 4.8 describes the use of packet numbers.

Payload: The payload of the packet.

The following packet types are defined:

| Type | Name | Section |
|---|---|---|
| 0x7F | Initial | Section 4.4.1 |
| 0x7E | Retry | Section 4.4.2 |
| 0x7D | Handshake | Section 4.4.3 |
| 0x7C | 0-RTT Protected | Section 4.5 |

Table 1: Long Header Packet Types

The header form, type, connection ID lengths octet, destination and source connection IDs, and version fields of a long header packet are version-independent. The packet number and values for packet types defined in Table 1 are version-specific. See [QUIC-INVARIANTS] for details on how packets from different versions of QUIC are interpreted.

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

End of the Payload field (which is also the end of the long header packet) is determined by the value of the Payload Length field. Senders can coalesce multiple long header packets into one UDP datagram. See Section 4.6 for more details.

## 4.2. Short Header

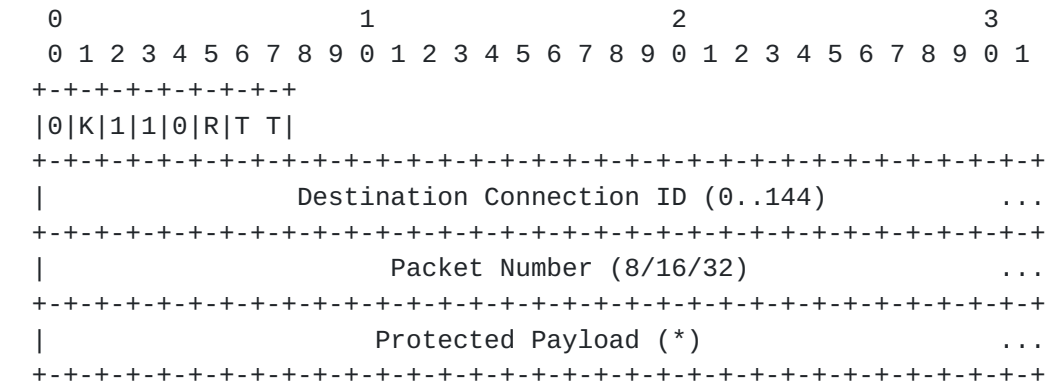



Figure 2: Short Header Format

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.

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

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

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

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

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

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

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

Reserved: The sixth bit (0x4) of octet 0 is reserved for experimentation.

Short Packet Type: The remaining 2 bits of octet 0 include one of 4 packet types. Table 2 lists the types that are defined for short packets.

Destination Connection ID: The Destination Connection ID is a connection ID that is chosen by the intended recipient of the packet. See Section 4.7 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 |
|---|---|
| 0x0 | 1 octet |
| 0x1 | 2 octets |
| 0x2 | 4 octets |

Table 2: Short Header Packet Types

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

## 4.3. Version Negotiation Packet

A Version Negotiation packet is inherently not version-specific, and does not use the long packet header (see Section 4.1. Upon receipt by a client, it will appear to be a packet using the long header, but will be identified as a Version Negotiation packet based on the Version field having a value of 0.

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

The layout of a Version Negotiation packet is:

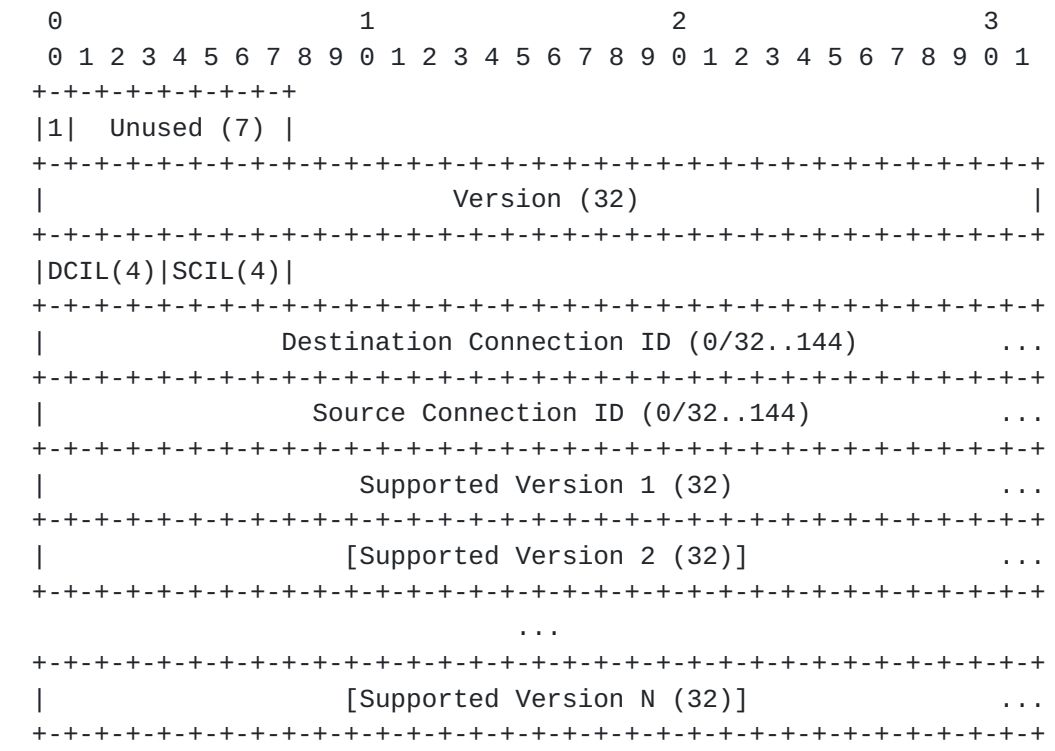



Figure 3: Version Negotiation Packet

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

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

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

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

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

The Version Negotiation packet does not include the Packet Number and Length fields present in other packets that use the long header form.

Consequently, a Version Negotiation packet consumes an entire UDP datagram.

See Section 6.2 for a description of the version negotiation process.

## 4.4. Cryptographic Handshake Packets

Once version negotiation is complete, the cryptographic handshake is used to agree on cryptographic keys. The cryptographic handshake is carried in Initial (Section 4.4.1), Retry (Section 4.4.2) and Handshake (Section 4.4.3) packets.

All these packets use the long header and contain the current QUIC version in the version field.

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

### 4.4.1. Initial Packet

The Initial packet uses long headers with a type value of 0x7F. It carries the first cryptographic handshake message sent by the client.

If the client has not previously received a Retry packet from the server, it populates the Destination Connection ID field with a randomly selected value. This MUST be at least 8 octets in length. Until a packet is received from the server, the client MUST use the same random value unless it also changes the Source Connection ID (which effectively starts a new connection attempt). The randomized Destination Connection ID is used to determine packet protection keys, but is not included in server packets.

If the client received a Retry packet and is sending a second Initial packet, then it sets the Destination Connection ID to the value from the Source Connection ID in the Retry packet. Changing Destination Connection ID also results in a change to the keys used to protect the Initial packet.

The client populates the Source Connection ID field with a value of its choosing and sets the low bits of the ConnID Len field to match.

The first Initial packet that is sent by a client contains a randomized packet number. All subsequent packets contain a packet number that is incremented by one, see (Section 4.8).

The payload of an Initial packet conveys a STREAM frame (or frames) for stream 0 containing a cryptographic handshake message. The stream in this packet always starts at an offset of 0 (see Section 6.5) and the complete cryptographic handshake message MUST fit in a single packet (see Section 6.3).

The payload of a UDP datagram carrying the Initial packet MUST be expanded to at least 1200 octets (see Section 8), by adding PADDING frames to the Initial packet and/or by combining the Initial packet with a 0-RTT packet (see Section 4.6).

The client uses the Initial packet type for any packet that contains an initial cryptographic handshake message. This includes all cases where a new packet containing the initial cryptographic message needs to be created, this includes the packets sent after receiving a Version Negotiation (Section 4.3) or Retry packet (Section 4.4.2).

### 4.4.2. Retry Packet

A Retry packet uses long headers with a type value of 0x7E. It carries cryptographic handshake messages and acknowledgments. It is used by a server that wishes to perform a stateless retry (see Section 6.5).

The server populates the Destination Connection ID with the connection ID that the client included in the Source Connection ID of the Initial packet. This might be a zero-length value.

The server includes a connection ID of its choice in the Source Connection ID field. The client MUST use this connection ID in the Destination Connection ID of subsequent packets that it sends.

The packet number field echoes the packet number field from the triggering client packet.

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

After receiving a Retry packet, the client uses a new Initial packet containing the next cryptographic handshake message. The client retains the state of its cryptographic handshake, but discards all transport state. The Initial packet that is generated in response to a Retry packet includes STREAM frames on stream 0 that start again at an 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 6.2). 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 Retry packet contains at least two frames. It MUST include a STREAM frame on stream 0 with offset 0 containing the server's cryptographic stateless retry material. It MUST also include an ACK frame to acknowledge the client's Initial packet. It MAY additionally include PADDING frames. The next STREAM frame sent by the server will also start at stream offset 0.

### 4.4.3. Handshake Packet

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

The Destination Connection ID field in a Handshake packet contains a connection ID that is chosen by the recipient of the packet; the Source Connection ID includes the connection ID that the sender of the packet wishes to use (see Section 4.7).

The first Handshake packet sent by a server contains a randomized packet number. This value is increased for each subsequent packet sent by the server as described in Section 4.8. The client increments the packet number from its previous packet by one for each Handshake packet that it sends (which might be an Initial, 0-RTT Protected, or Handshake packet).

Servers MUST NOT send more than three Handshake packets without receiving a packet from a verified source address. Source addresses can be verified through an address validation token, receipt of the final cryptographic message from the client, or by receiving a valid PATH_RESPONSE frame from the client.

If the server expects to generate more than three Handshake packets in response to an Initial packet, it SHOULD include a PATH_CHALLENGE frame in each Handshake packet that it sends. After receiving at least one valid PATH_RESPONSE frame, the server can send its remaining Handshake packets. Servers can instead perform address validation using a Retry packet; this requires less state on the server, but could involve additional computational effort depending on implementation choices.

The payload of this packet contains STREAM frames and could contain PADDING, ACK, PATH_CHALLENGE, or PATH_RESPONSE frames. Handshake

packets MAY contain CONNECTION_CLOSE frames if the handshake is unsuccessful.

## 4.5. Protected Packets

Packets that are protected with 0-RTT keys are sent with long headers; all packets protected with 1-RTT keys are sent with short 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 0x7C. The connection ID fields for a 0-RTT packet MUST match the values used in the Initial packet (Section 4.4.1).

The client can send 0-RTT packets after receiving a Handshake packet (Section 4.4.3), if that packet does not complete the handshake. Even if the client receives a different connection ID in the Handshake packet, it MUST continue to use the same Destination Connection ID for 0-RTT packets, see Section 4.7.

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 4.8 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 5.

## 4.6. Coaslescing Packets

A sender can coalesce multiple QUIC packets (typically a Cryptographic Handshake packet and a Protected packet) into one UDP datagram. This can reduce the number of UDP datagrams needed to send application data during the handshake and immediately afterwards. A packet with a short header does not include a length, so it has to be the last packet included in a UDP datagram.

The sender MUST NOT coalesce QUIC packets belonging to different QUIC connections into a single UDP datagram.

Every QUIC packet that is coalesced into a single UDP datagram is separate and complete. Though the values of some fields in the packet header might be redundant, no fields are omitted. The receiver of coalesced QUIC packets MUST individually process each

QUIC packet and separately acknowledge them, as if they were received as the payload of different UDP datagrams.

### 4.7. Connection ID

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

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

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

Short headers only include the Destination Connection ID and omit the explicit length. The length of the Destination Connection ID field is expected to be known to endpoints.

Endpoints using a connection-ID based load balancer could agree with the load balancer on a fixed or minimum length and on an encoding for connection IDs. This fixed portion could encode an explicit length, which allows the entire connection ID to vary in length and still be used by the load balancer.

The very first packet sent by a client includes a random value for Destination Connection ID. The same value MUST be used for all 0-RTT packets sent on that connection (Section 4.5). This randomized value is used to determine the handshake packet protection keys (see Section 5.3.2 of [QUIC-TLS]).

A Version Negotiation (Section 4.3) packet MUST use both connection IDs selected by the client, swapped to ensure correct routing toward the client.

The connection ID can change over the lifetime of a connection, especially in response to connection migration (Section 6.8). NEW_CONNECTION_ID frames (Section 7.13) are used to provide new connection ID values.

## 4.8. Packet Numbers

The packet number is an integer in the range 0 to 2^62-1. The value is used in determining the 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 4.8.1).

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 2^62 - 1, the sender MUST close the connection without sending a CONNECTION_CLOSE frame or any further packets; a server MAY send a Stateless Reset (Section 6.9.4) in response to further packets that it receives.

For the packet header, the number of bits required to represent the packet number are reduced by including only the least significant bits of the packet number. 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.

A Version Negotiation packet (Section 4.3) does not include a packet number. The Retry packet (Section 4.4.2) has special rules for populating the packet number field.

### 4.8.1. Initial Packet Number

The initial value for packet number MUST be selected randomly from a range between 0 and 2^32 - 1025 (inclusive). This value is selected so that Initial and Handshake packets exercise as many possible values for the Packet Number field as possible.

Limiting the range allows both for loss of packets and for any stateless exchanges. Packet numbers are incremented for subsequent packets, but packet loss and stateless handling can both mean that the first packet sent by an endpoint isn't necessarily the first packet received by its peer. The first packet received by a peer cannot be 2^32 or greater or the recipient will incorrectly assume a packet number that is 2^32 values lower and discard the packet.

Use of a secure random number generator [RFC4086] is not necessary for generating the initial packet number, nor is it necessary that the value be uniformly distributed.

## 5. Frames and Frame Types

The payload of all packets, after removing packet protection, consists of a sequence of frames, as shown in Figure 4. Version Negotiation and Stateless Reset do not contain frames.

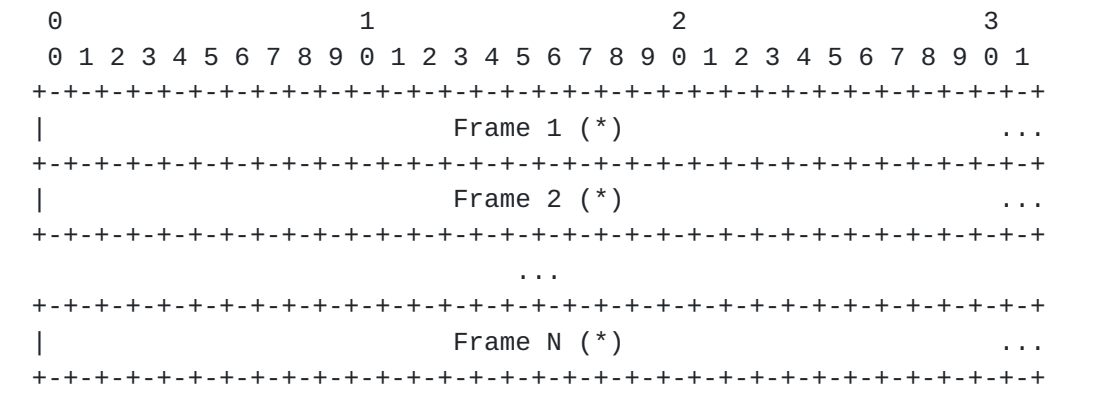



Figure 4: Contents of Protected Payload

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:

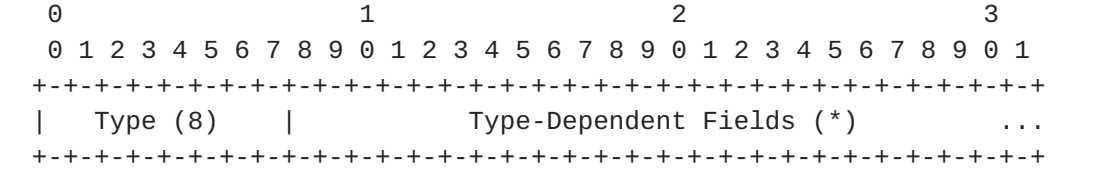



Figure 5: Generic Frame Layout

Frame types are listed in Table 3. Note that the Frame Type byte in STREAM 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 7.2 |
| 0x01 | RST_STREAM | Section 7.3 |
| 0x02 | CONNECTION_CLOSE | Section 7.4 |
| 0x03 | APPLICATION_CLOSE | Section 7.5 |
| 0x04 | MAX_DATA | Section 7.6 |
| 0x05 | MAX_STREAM_DATA | Section 7.7 |
| 0x06 | MAX_STREAM_ID | Section 7.8 |
| 0x07 | PING | Section 7.9 |
| 0x08 | BLOCKED | Section 7.10 |
| 0x09 | STREAM_BLOCKED | Section 7.11 |
| 0x0a | STREAM_ID_BLOCKED | Section 7.12 |
| 0x0b | NEW_CONNECTION_ID | Section 7.13 |
| 0x0c | STOP_SENDING | Section 7.14 |
| 0x0d | ACK | Section 7.15 |
| 0x0e | PATH_CHALLENGE | Section 7.16 |
| 0x0f | PATH_RESPONSE | Section 7.17 |
| 0x10 - 0x17 | STREAM | Section 7.18 |

Table 3: Frame Types

## 6. 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 6.3. 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 6.8. Finally a connection may be terminated by either endpoint, as described in Section 6.9.

## 6.1. Matching Packets to Connections

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

Hosts try to associate a packet with an existing connection. If the packet has a Destination Connection ID corresponding to an existing connection, QUIC processes that packet accordingly. Note that a NEW_CONNECTION_ID frame (Section 7.13) would associate more than one connection ID with a connection.

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

### 6.1.1. Client Packet Handling

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

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

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

### 6.1.2. Server Packet Handling

If a server receives a packet that has an unsupported version and sufficient length to be an Initial packet for some version supported by the server, it SHOULD send a Version Negotiation packet as described in Section 6.2.1. Servers MAY rate control these packets to avoid storms of Version Negotiation packets.

The first packet for an unsupported version can use different semantics and encodings for any version-specific field. In

particular, different packet protection keys might be used for different versions. Servers that do not support a particular version are unlikely to be able to decrypt the content of the packet. Servers SHOULD NOT attempt to decode or decrypt a packet from an unknown version, but instead send a Version Negotiation packet, provided that the packet is sufficiently long.

Servers MUST drop other packets that contain unsupported versions.

Packets with a supported version, or no version field, are matched to a connection as described in Section 6.1. If not matched, the server continues below.

If the packet is an Initial packet fully conforming with the specification, the server proceeds with the handshake (Section 6.3). This commits the server to the version that the client selected.

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

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

Servers MUST drop incoming packets under all other circumstances. They SHOULD send a Stateless Reset (Section 6.9.4) if a connection ID is present in the header.

## 6.2. Version Negotiation

Version negotiation ensures that client and server agree to a QUIC version that is mutually supported. A server sends a Version Negotiation packet in response to each packet that might initiate a new connection, see Section 6.1 for details.

The size of the first packet sent by a client will determine whether a server sends a Version Negotiation packet. Clients that support multiple QUIC versions SHOULD pad their Initial packets to reflect the largest minimum Initial packet size of all their versions. This ensures that that the server responds if there are any mutually supported versions.

### 6.2.1. Sending Version Negotiation Packets

If the version selected by the client is not acceptable to the server, the server responds with a Version Negotiation packet (see Section 4.3). This includes a list of versions that the server will accept.

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

### 6.2.2. Handling Version Negotiation Packets

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

Once the Version Negotiation packet is determined to be valid, the client then selects an acceptable protocol version from the list provided by the server. The client then attempts to create a connection using that version. Though the contents of the Initial packet the client sends might not change in response to version negotiation, a client MUST increase the packet number it uses on every packet it sends. Packets MUST continue to use long headers and MUST include the new negotiated protocol version.

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

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

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

Version negotiation packets have no cryptographic protection. The result of the negotiation MUST be revalidated as part of the cryptographic handshake (see Section 6.4.4).

### 6.2.3. Using Reserved Versions

For a server to use a new version in the future, clients must correctly handle unsupported versions. To help ensure this, a server SHOULD include a reserved version (see Section 3) while generating a Version Negotiation packet.

The design of version negotiation permits a server to avoid maintaining state for packets that it rejects in this fashion. The validation of version negotiation (see Section 6.4.4) only validates the result of version negotiation, which is the same no matter which reserved version was sent. A server MAY therefore send different reserved version numbers in the Version Negotiation Packet and in its transport parameters.

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

## 6.3. 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 6.4)
- authenticated confirmation of version negotiation (see Section 6.4.4)

- 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 6.6)

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].

### 6.4. 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_bidi(2),
   idle_timeout(3),
   max_packet_size(5),
   stateless_reset_token(6),
   ack_delay_exponent(7),
   initial_max_stream_id_uni(8),
   (65535)
} TransportParameterId;

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

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

      case encrypted_extensions:
         QuicVersion negotiated_version;
         QuicVersion supported_versions<4..2^8-4>;
   };
   TransportParameter parameters<22..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 6.4.4) before the connection establishment is considered properly complete.

Definitions for each of the defined transport parameters are included in Section 6.4.1. Any given parameter MUST appear at most once in a given transport parameters extension. An endpoint MUST treat receipt

of duplicate transport parameters as a connection error of type TRANSPORT_PARAMETER_ERROR.

### 6.4.1. 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 7.7) 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 octets. This is equivalent to sending a MAX_DATA (Section 7.6) for the connection 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).

An endpoint MAY use the following transport parameters:

initial_max_streams_bidi (0x0002): The initial maximum bidirectional streams parameter contains the initial maximum number of application-owned bidirectional streams the peer may initiate, encoded as an unsigned 16-bit integer. If this parameter is absent or zero, application-owned bidirectional streams cannot be created until a MAX_STREAM_ID frame is sent. Note that a value of 0 does not prevent the cryptographic handshake stream (that is, stream 0) from being used. Setting this parameter is equivalent to sending a MAX_STREAM_ID (Section 7.8) immediately after completing the handshake containing the corresponding Stream ID. For example, a value of 0x05 would be equivalent to receiving a MAX_STREAM_ID containing 20 when received by a client or 17 when received by a server.

initial_max_stream_id_uni (0x0008): The initial maximum unidirectional streams parameter contains the initial maximum number of application-owned unidirectional streams the peer may initiate, encoded as an unsigned 16-bit integer. If this parameter is absent or zero, unidirectional streams cannot be created until a MAX_STREAM_ID frame is sent. Setting this

parameter is equivalent to sending a MAX_STREAM_ID (Section 7.8) immediately after completing the handshake containing the corresponding Stream ID. For example, a value of 0x05 would be equivalent to receiving a MAX_STREAM_ID containing 18 when received by a client or 19 when received by a server.

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 4.5).

ack_delay_exponent (0x0007): An 8-bit unsigned integer value indicating an exponent used to decode the ACK Delay field in the ACK frame, see Section 7.15. If this value is absent, a default value of 3 is assumed (indicating a multiplier of 8). The default value is also used for ACK frames that are sent in Initial, Handshake, and Retry packets. Values above 20 are invalid.

A server MAY include the following transport parameters:

stateless_reset_token (0x0006): The Stateless Reset Token is used in verifying a stateless reset, see Section 6.9.4. This parameter is a sequence of 16 octets.

A client MUST NOT include a stateless reset token. A server MUST treat receipt of a stateless_reset_token transport parameter as a connection error of type TRANSPORT_PARAMETER_ERROR.

### 6.4.2. Values of Transport Parameters for 0-RTT

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

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

A server MAY accept 0-RTT and subsequently provide different values for transport parameters for use in the new connection. If 0-RTT

data is accepted by the server, the server MUST NOT reduce any limits or alter any values that might be violated by the client with its 0-RTT data. In particular, a server that accepts 0-RTT data MUST NOT set values for initial_max_data 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_bidi or initial_max_stream_id_uni.

Omitting or setting a zero value for certain transport parameters can result in 0-RTT data being enabled, but not usable. The following transport parameters SHOULD be set to non-zero values for 0-RTT: initial_max_stream_id_bidi, initial_max_stream_id_uni, initial_max_data, initial_max_stream_data.

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

### 6.4.3. 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 13.1.

### 6.4.4. Version Negotiation Validation

Though the cryptographic handshake has integrity protection, two forms of QUIC version downgrade are possible. In the first, an attacker replaces the QUIC version in the Initial packet. In the second, a fake Version Negotiation packet is sent by an attacker. To protect against these attacks, the transport parameters include three fields that encode version information. These parameters are used to retroactively authenticate the choice of version (see Section 6.2).

The cryptographic handshake provides integrity protection for the negotiated version as part of the transport parameters (see Section 6.4). As a result, attacks on version negotiation by an attacker can be detected.

The client includes the initial_version field in its transport parameters. The initial_version is the version that the client initially attempted to use. If the server did not send a Version Negotiation packet Section 4.3, this will be identical to the negotiated_version field in the server transport parameters.

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

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

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

The server includes both the version of QUIC that is in use and a list of the QUIC versions that the server supports.

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

The server includes a list of versions that it would send in any version negotiation packet (Section 4.3) in the supported_versions field. The server populates this field even if it did not send a version negotiation packet.

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

When an endpoint accepts multiple QUIC versions, it can potentially interpret transport parameters as they are defined by any of the QUIC versions it supports. The version field in the QUIC packet header is authenticated using transport parameters. The position and the format of the version fields in transport parameters MUST either be identical across different QUIC versions, or be unambiguously

different to ensure no confusion about their interpretation. One way that a new format could be introduced is to define a TLS extension with a different codepoint.

## 6.5. 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 6.6, or to defer connection establishment costs.

A server that generates a response to an initial packet without retaining connection state MUST use the Retry packet (Section 4.4.2). 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 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 Retry packet type is used to carry the HelloRetryRequest message.

## 6.6. 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 1200 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.

A different type of source address validation is performed after a connection migration, see Section 6.7.

### 6.6.1. 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 6.6.3), 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.

### 6.6.2. 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 6.8). 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.

### 6.6.3. 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.

### 6.7. Path Validation

Path validation is used by an endpoint to verify reachability of a peer over a specific path. That is, it tests reachability between a specific local address and a specific peer address, where an address is the two-tuple of IP address and port. Path validation tests that packets can be both sent to and received from a peer.

Path validation is used during connection migration (see Section 6.8) by the migrating endpoint to verify reachability of a peer from a new local address. Path validation is also used by the peer to verify that the migrating endpoint is able to receive packets sent to its new address. That is, that the packets received from the migrating endpoint do not carry a spoofed source address.

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

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

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

#### 6.7.1. Initiation

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

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

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

### 6.7.2. Response

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

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

### 6.7.3. Completion

A new address is considered valid when a PATH_RESPONSE frame is received containing data that was sent in a previous PATH_CHALLENGE. Receipt of an acknowledgment for a packet containing a PATH_CHALLENGE frame is not adequate validation, since the acknowledgment can be spoofed by a malicious peer.

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

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

### 6.7.4. Abandonment

An endpoint SHOULD abandon path validation after sending some number of PATH_CHALLENGE frames or after some time has passed. When setting this timer, implementations are cautioned that the new path could have a longer round-trip time than the original.

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

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

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

## 6.8. Connection Migration

QUIC allows connections to survive changes to endpoint addresses (that is, IP address and/or port), such as those caused by a endpoint migrating to a new network. This section describes the process by which an endpoint migrates to a new address.

An endpoint MUST NOT initiate connection migration before the handshake is finished and the endpoint has 1-RTT keys.

This document limits migration of connections to new client addresses. Clients are responsible for initiating all migrations. Servers do not send non-probing packets (see Section 6.8.1) toward a client address until it sees a non-probing packet from that address. If a client receives packets from an unknown server address, the client MAY discard these packets. Migrating a connection to a new server address is left for future work.

### 6.8.1. Probing a New Path

An endpoint MAY probe for peer reachability from a new local address using path validation Section 6.7 prior to migrating the connection to the new local address. Failure of path validation simply means that the new path is not usable for this connection. Failure to validate a path does not cause the connection to end unless there are no valid alternative paths available.

An endpoint uses a new connection ID for probes sent from a new local address, see Section 6.8.5 for further discussion.

Receiving a PATH_CHALLENGE frame from a peer indicates that the peer is probing for reachability on a path. An endpoint sends a PATH_RESPONSE in response as per Section 6.7.

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

### 6.8.2. Initiating Connection Migration

A endpoint can migrate a connection to a new local address by sending packets containing frames other than probing frames from that address.

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

When migrating, the new path might not support the endpoint's current sending rate. Therefore, the endpoint resets its congestion controller, as described in Section 6.8.4.

Receiving acknowledgments for data sent on the new path serves as proof of the peer's reachability from the new address. Note that since acknowledgments may be received on any path, return reachability on the new path is not established. To establish return reachability on the new path, an endpoint MAY concurrently initiate path validation Section 6.7 on the new path.

### 6.8.3. Responding to Connection Migration

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

In response to such a packet, an endpoint MUST start sending subsequent packets to the new peer address and MUST initiate path validation (Section 6.7) to verify the peer's ownership of the unvalidated address.

An endpoint MAY send data to an unvalidated peer address, but it MUST protect against potential attacks as described in Section 6.8.3.1 and

Section 6.8.3.2. An endpoint MAY skip validation of a peer address if that address has been seen recently.

An endpoint only changes the address that it sends packets to in response to the highest-numbered non-probing packet. This ensures that an endpoint does not send packets to an old peer address in the case that it receives reordered packets.

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

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

After verifying a new client address, the server SHOULD send new address validation tokens (Section 6.6) to the client.

#### 6.8.3.1. Handling Address Spoofing by a Peer

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

As described in Section 6.8.3, an endpoint is required to validate a peer's new address to confirm the peer's possession of the new address. Until a peer's address is deemed valid, an endpoint MUST limit the rate at which it sends data to this address. The endpoint MUST NOT send more than a minimum congestion window's worth of data per estimated round-trip time (kMinimumWindow, as defined in [QUIC-RECOVERY]). In the absence of this limit, an endpoint risks being used for a denial of service attack against an unsuspecting victim. Note that since the endpoint will not have any round-trip time measurements to this address, the estimate SHOULD be the default initial value (see [QUIC-RECOVERY]).

If an endpoint skips validation of a peer address as described in Section 6.8.3, it does not need to limit its sending rate.

#### 6.8.3.2. Handling Address Spoofing by an On-path Attacker

An on-path attacker could cause a spurious connection migration by copying and forwarding a packet with a spoofed address such that it arrives before the original packet. The packet with the spoofed address will be seen to come from a migrating connection, and the original packet will be seen as a duplicate and dropped. After a spurious migration, validation of the source address will fail

because the entity at the source address does not have the necessary cryptographic keys to read or respond to the PATH_CHALLENGE frame that is sent to it even if it wanted to.

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

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

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

#### 6.8.4. Loss Detection and Congestion Control

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

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

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

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

While multiple paths might be used during connection migration, a single congestion control context and a single loss recovery context (as described in [QUIC-RECOVERY]) may be adequate. A sender can make

exceptions for probe packets so that their loss detection is independent and does not unduly cause the congestion controller to reduce its sending rate. An endpoint might arm a separate alarm when a PATH_CHALLENGE is sent, which is disarmed when the corresponding PATH_RESPONSE is received. If the alarm fires before the PATH_RESPONSE is received, the endpoint might send a new PATH_CHALLENGE, and restart the alarm for a longer period of time.

### 6.8.5. Privacy Implications of Connection Migration

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

An endpoint might need to send packets on multiple networks without receiving any response from its peer. To ensure that the endpoint is not linkable across each of these changes, a new connection ID and packet number gap are needed for each network. To support this, each endpoint 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.

An endpoint that does not require the use of a connection ID should not request that its peer use a connection ID. Such an endpoint does not need to provide new connection IDs using the NEW_CONNECTION_ID frame.

An endpoint which wishes to break linkability upon changing networks MUST use the connection ID provided by its peer as well as incrementing the packet sequence number by an externally unpredictable value computed as described in Section 6.8.5.1. Packet number gaps are cumulative. An endpoint 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.

An endpoint 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. An endpoint MUST discard packets that contain a smaller gap than it advertised.

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

A client might wish to reduce linkability by employing a new connection ID and source UDP port when sending traffic after a period of inactivity. Changing the UDP port from which it sends packets at the same time might cause the packet to appear as a connection migration. This ensures that the mechanisms that support migration are exercised even for clients that don't experience NAT rebindings or genuine migrations. Changing port number can cause a peer to reset its congestion state (see Section 6.8.4), so the port SHOULD only be changed infrequently.

An endpoint that receives a successfully authenticated packet with a previously unused connection ID MUST use the next available connection ID for any packets it sends to that address. To avoid changing connection IDs multiple times when packets arrive out of order, endpoints MUST change only in response to a packet that increases the largest received packet number. Failing to do this could allow for use of that connection ID to link activity on new paths. There is no need to move to a new connection ID if the address of a peer changes without also changing the connection ID.

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.

#### 6.8.5.1. 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 Section 5.6 of [QUIC-TLS].

### 6.9. Connection Termination

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

- idle timeout (Section 6.9.2)

- immediate close (Section 6.9.3)

- stateless reset (Section 6.9.4)

### 6.9.1. Closing and Draining Connection States

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

An endpoint enters a closing period after initiating an immediate close (Section 6.9.3). While closing, an endpoint MUST NOT send packets unless they contain a CONNECTION_CLOSE or APPLICATION_CLOSE frame (see Section 6.9.3 for details).

In the closing state, only a packet containing a closing frame can be sent. An endpoint retains only enough information to generate a packet containing a closing frame and to identify packets as belonging to the connection. The connection ID and QUIC version is sufficient information to identify packets for a closing connection; an endpoint can discard all other connection state. An endpoint MAY retain packet protection keys for incoming packets to allow it to read and process a closing frame.

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

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

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

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

The draining and closing periods do not apply when a stateless reset (Section 6.9.4) is sent.

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

An endpoint could receive packets from a new source address, indicating a client connection migration (Section 6.8), while in the closing period. An endpoint in the closing state MUST strictly limit the number of packets it sends to this new address until the address is validated (see Section 6.7). A server in the closing state MAY instead choose to discard packets received from a new source address.

### 6.9.2. Idle Timeout

A connection that remains idle for longer than the idle timeout (see Section 6.4.1) is closed. A connection enters the draining state when the idle timeout expires.

The time at which an idle timeout takes effect won't be perfectly synchronized on both endpoints. An endpoint that sends packets near the end of an idle period could have those packets discarded if its peer enters the draining state before the packet is received.

### 6.9.3. Immediate Close

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

After sending a closing frame, endpoints immediately enter the closing state. During the closing period, an endpoint that sends a closing frame SHOULD respond to any packet that it receives with another packet containing a closing frame. To minimize the state that an endpoint maintains for a closing connection, endpoints MAY send the exact same packet. However, endpoints SHOULD limit the number of packets they generate containing a closing frame. For instance, an endpoint could progressively increase the number of packets that it receives before sending additional packets or increase the time between packets.

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

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

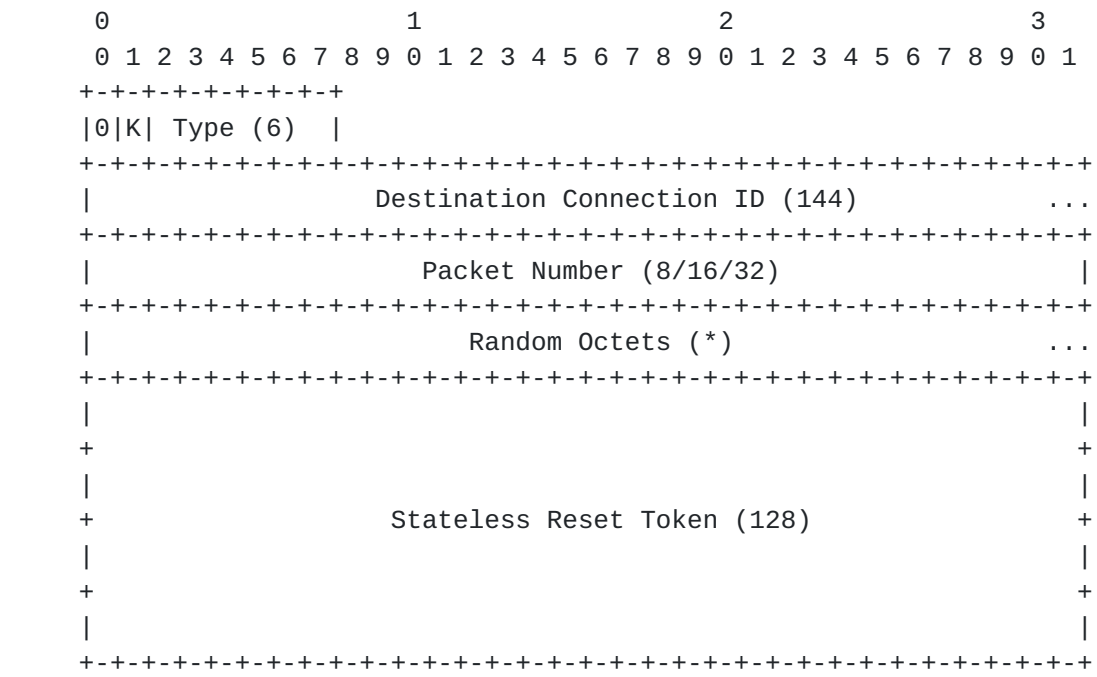
An immediate close can be used after an application protocol has arranged to close a connection. This might be after the application protocols negotiates a graceful shutdown. The application protocol exchanges whatever messages that are needed to cause both endpoints to agree to close the connection,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

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

A server generates a random 18-octet Destination Connection ID field. For a client that depends on the server including a connection ID, this will mean that this value differs from previous packets. Ths results in two problems:

- The packet might not reach the client. If the Destination Connection ID is critical for routing toward the client, then this packet could be incorrectly routed. This causes the stateless reset to be ineffective in causing errors to be quickly detected and recovered. In this case, clients will need to rely on other methods - such as timers - to detect that the connection has failed.

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

The Packet Number field is set to a randomized value. The server SHOULD send a packet with a short header and a type of 0x1F. This produces the shortest possible packet number encoding, which minimizes the perceived gap between the last packet that the server sent and this packet. A server MAY use a different short header type, indicating a different packet number length, but a longer

packet number encoding might allow this message to be identified as a stateless reset more easily using heuristics.

After the Packet Number, the server pads the message with an arbitrary number of octets containing random values.

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

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

This stateless reset design is specific to QUIC version 1. A server that supports multiple versions of QUIC needs to generate a stateless reset that will be accepted by clients that support any version that the server might support (or might have supported prior to losing state). Designers of new versions of QUIC need to be aware of this and either reuse this design, or use a portion of the packet other than the last 16 octets for carrying data.

#### 6.9.4.1. Detecting a Stateless Reset

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

#### 6.9.4.2. 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, the server's connection ID (see Section 4.7), 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 use a zero-length connection ID.

Revealing the Stateless Reset Token allows any entity to terminate the connection, so a value can only be used once. This method for choosing the Stateless Reset Token means that the combination of 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.

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

## 7. Frame Types and Formats

As described in Section 5, packets contain one or more frames. This section describes the format and semantics of the core QUIC frame types.

### 7.1. Variable-Length Integer Encoding

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

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

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

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

Table 4: Summary of Integer Encodings

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

Error codes (Section 11.3) are described using integers, but do not use this encoding.

## 7.2. 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.

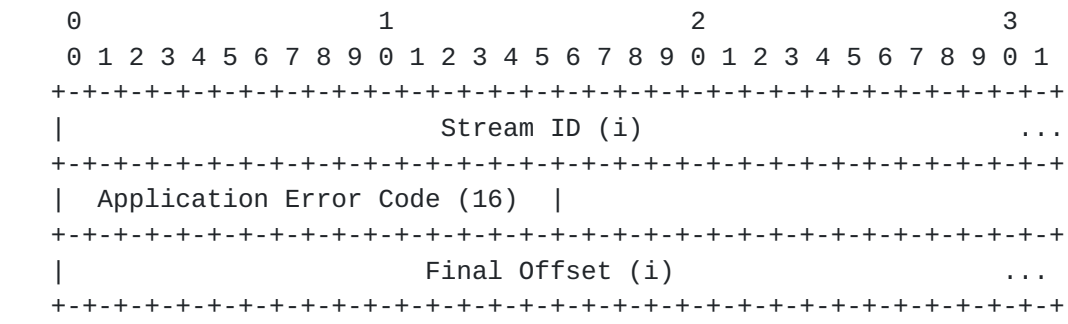
## 7.3. RST_STREAM Frame

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

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

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

The RST_STREAM frame is as follows:




The fields are:

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

Application Protocol Error Code: A 16-bit application protocol error code (see Section 11.4) which indicates why the stream is being closed.

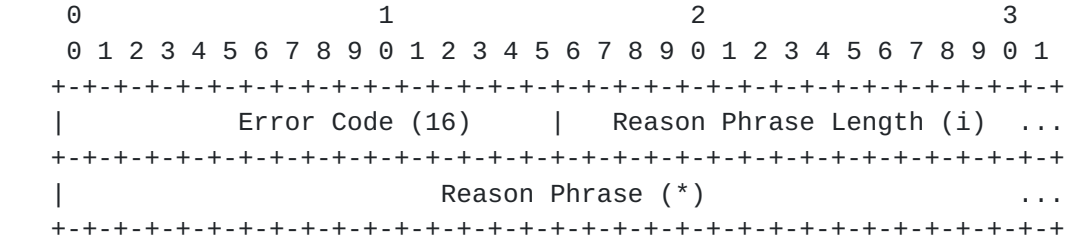
Final Offset: A variable-length integer indicating the absolute byte offset of the end of data written on this stream by the RST_STREAM sender.

## 7.4. CONNECTION_CLOSE frame

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

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

The CONNECTION_CLOSE frame is as follows:




The fields of a CONNECTION_CLOSE frame are as follows:

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

space defined in Section 11.3 (APPLICATION_CLOSE uses codes from the application protocol error code space, see Section 11.4).

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

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

## 7.5. APPLICATION_CLOSE frame

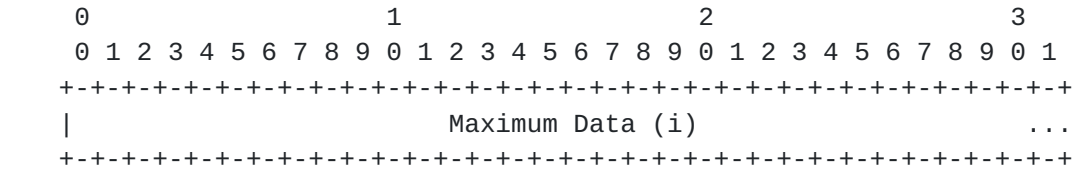
An APPLICATION_CLOSE frame (type=0x03) uses the same format as the CONNECTION_CLOSE frame (Section 7.4), except that it uses error codes from the application protocol error code space (Section 11.4) instead of the transport error code space.

Other than the error code space, the format and semantics of the APPLICATION_CLOSE frame are identical to the CONNECTION_CLOSE frame.

## 7.6. MAX_DATA Frame

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

The frame is as follows:




The fields in the MAX_DATA frame are as follows:

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

All data sent in STREAM frames counts toward this limit, with the exception of data on stream 0. The sum of the largest received offsets on all streams - including streams in terminal states, 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 6.4.2).

## 7.7. MAX_STREAM_DATA Frame

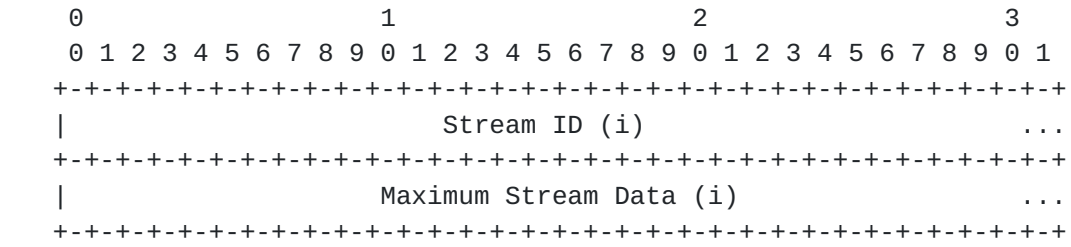
The MAX_STREAM_DATA frame (type=0x05) is used in flow control to inform a peer of the maximum amount of data that can be sent on a stream.

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

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

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

The frame is as follows:




The fields in the MAX_STREAM_DATA frame are as follows:

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

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

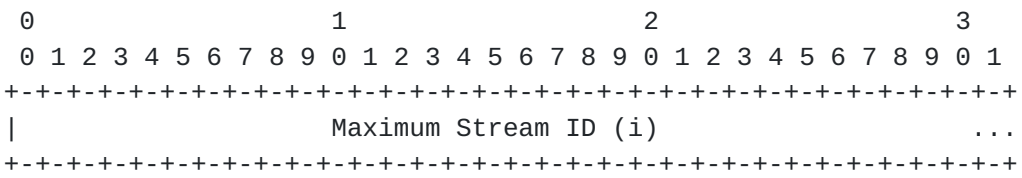
When counting data toward this limit, an endpoint accounts for the largest received offset of data that is sent or received on the stream. Loss or reordering can mean that the largest received offset on a stream can be greater than the total size of data received on that stream. Receiving STREAM frames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

The data sent on a stream MUST NOT exceed the largest maximum stream data value advertised by the receiver. An endpoint MUST terminate a connection with a FLOW_CONTROL_ERROR error if it receives more data than the largest maximum stream data that it has sent for the affected stream, unless this is a result of a change in the initial limits (see Section 6.4.2).

## 7.8. MAX_STREAM_ID Frame

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

The frame is as follows:




The fields in the MAX_STREAM_ID frame are as follows:

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

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

A peer MUST NOT initiate a stream with a higher stream ID than the greatest maximum stream ID it has received. An endpoint MUST terminate a connection with a STREAM_ID_ERROR error if a peer initiates a stream with a higher stream ID than it has sent, unless this is a result of a change in the initial limits (see Section 6.4.2).

## 7.9. PING Frame

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

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

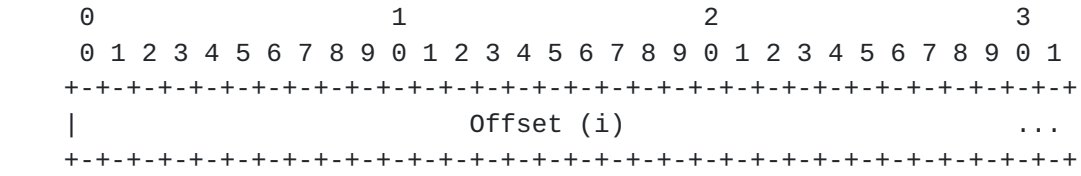
The PING frame can be used to keep a connection alive when an application or application protocol wishes to prevent the connection from timing out. An application protocol SHOULD provide guidance about the conditions under which generating a PING is recommended. This guidance SHOULD indicate whether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

A connection will time out if no packets are sent or received for a period longer than the time specified in the idle_timeout transport parameter (see Section 6.9). However, state in middleboxes might time out earlier than that. Though REQ-5 in [RFC4787] recommends a 2 minute timeout interval, experience shows that sending packets every 15 to 30 seconds is necessary to prevent the majority of middleboxes from losing state for UDP flows.

## 7.10. BLOCKED Frame

A sender SHOULD send a BLOCKED frame (type=0x08) when it wishes to send data, but is unable to due to connection-level flow control (see Section 10.2.1). BLOCKED frames can be used as input to tuning of flow control algorithms (see Section 10.1.2).

The BLOCKED frame is as follows:




The BLOCKED frame contains a single field.

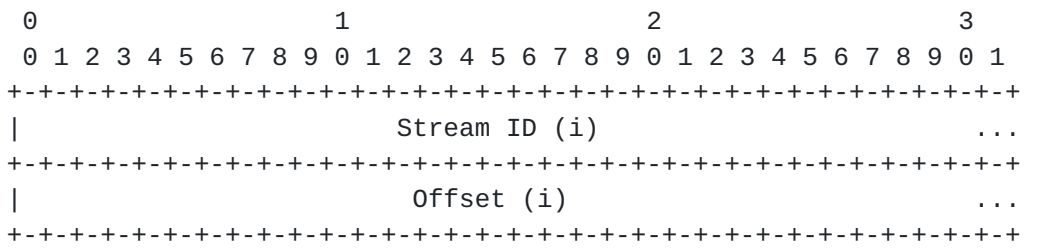
Offset: A variable-length integer indicating the connection-level offset at which the blocking occurred.

## 7.11. STREAM_BLOCKED Frame

A sender SHOULD send a STREAM_BLOCKED frame (type=0x09) when it wishes to send data, but is unable to due to stream-level flow control. This frame is analogous to BLOCKED (Section 7.10).

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

The STREAM_BLOCKED frame is as follows:




The STREAM_BLOCKED frame contains two fields:

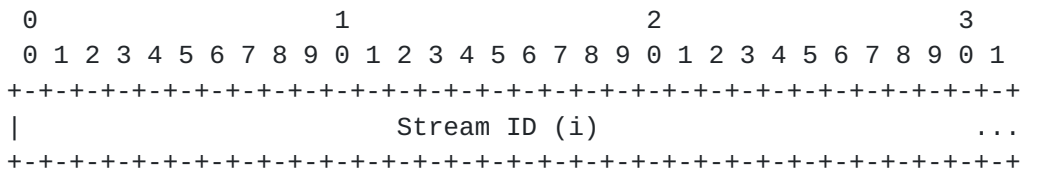
Stream ID: A variable-length integer indicating the stream which is flow control blocked.

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

## 7.12. STREAM_ID_BLOCKED Frame

A sender MAY send a STREAM_ID_BLOCKED frame (type=0x0a) when it wishes to open a stream, but is unable to due to the maximum stream ID limit set by its peer (see Section 7.8). This does not open the stream, but informs the peer that a new stream was needed, but the stream limit prevented the creation of the stream.

The STREAM_ID_BLOCKED frame is as follows:



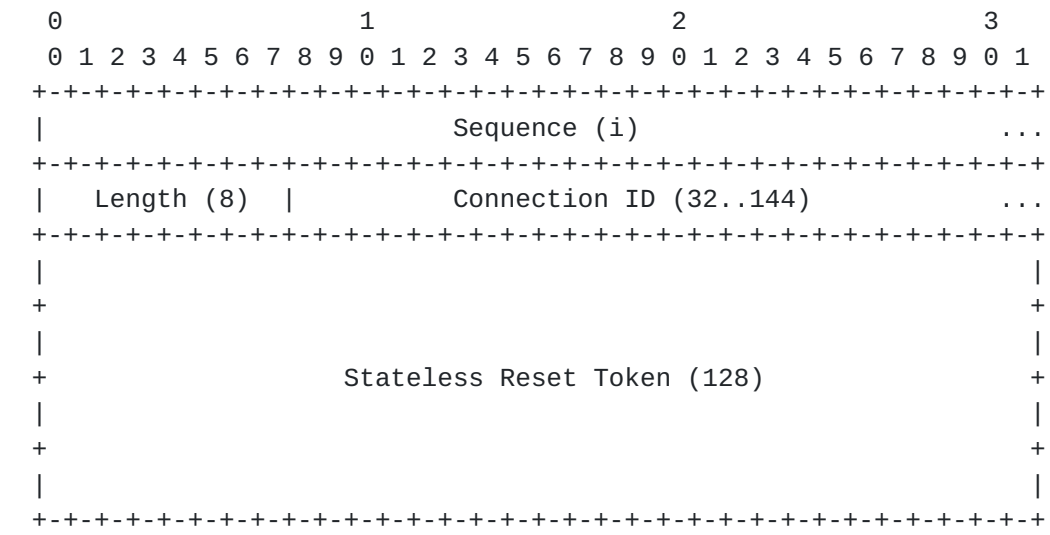

The STREAM_ID_BLOCKED frame contains a single field.

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

## 7.13. NEW_CONNECTION_ID Frame

An endpoint sends a NEW_CONNECTION_ID frame (type=0x0b) to provide its peer with alternative connection IDs that can be used to break linkability when migrating connections (see Section 6.8.5).

The NEW_CONNECTION_ID is as follows:




The fields are:

Sequence: A variable-length integer. This value starts at 0 and increases by 1 for each connection ID that is provided by the server. The connection ID that is assigned during the handshake is assumed to have a sequence of -1. That is, the value selected during the handshake comes immediately before the first value that a server can send.

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

Connection ID: A connection ID of the specified length.

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 6.9.4).

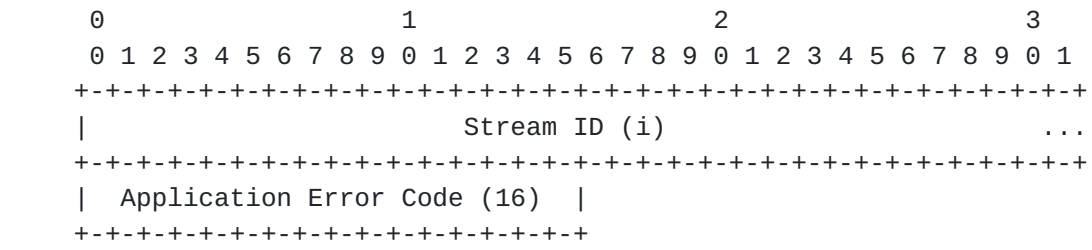
An endpoint MUST NOT send this frame if it currently requires that its peer send packets with a zero-length Destination Connection ID. Changing the length of a connection ID to or from zero-length makes it difficult to identify when the value of the connection ID changed. An endpoint that is sending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

### 7.14. STOP_SENDING Frame

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

Receipt of a STOP_SENDING frame is only valid for a send stream that exists and is not in the "Ready" state (see Section 9.2.1). Receiving a STOP_SENDING frame for a send stream that is "Ready" or non-existent MUST be treated as a connection error of type PROTOCOL_VIOLATION. An endpoint that receives a STOP_SENDING frame for a receive-only stream MUST terminate the connection with error PROTOCOL_VIOLATION.

The STOP_SENDING frame is as follows:




The fields are:

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

Application Error Code: A 16-bit, application-specified reason the sender is ignoring the stream (see Section 11.4).

### 7.15. ACK Frame

Receivers send ACK frames (type=0x0d) to inform senders which packets they have received and processed. A sent packet that has never been acknowledged is missing. The ACK frame contains any number of ACK blocks. ACK blocks are ranges of acknowledged packets.

Unlike TCP SACKs, QUIC acknowledgements 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 Retry packets. Retry packets include the packet number from the Initial packet it responds to. Version Negotiation packets cannot be acknowledged because they do not contain a packet number. Rather than relying on ACK frames, these

packets are implicitly acknowledged by the next Initial packet sent by the client.

An ACK frame is shown below.

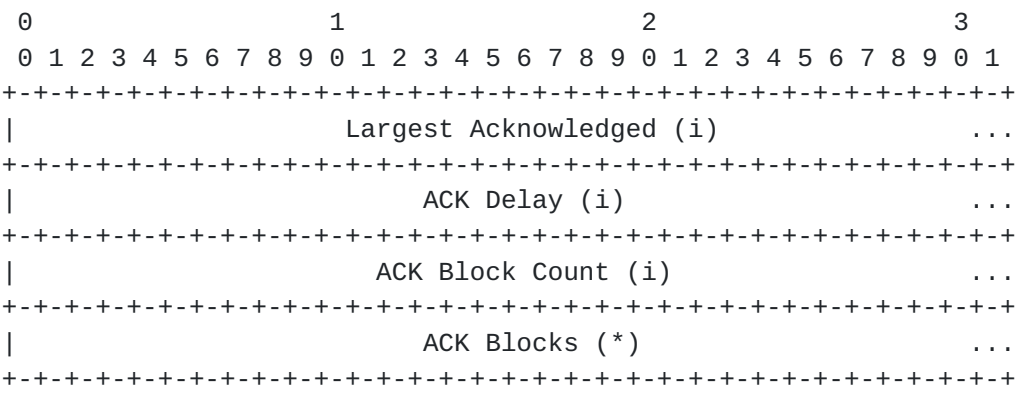



Figure 7: ACK Frame Format

The fields in the ACK frame are as follows:

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

ACK Delay: A variable-length integer including the time in microseconds that the largest acknowledged packet, as indicated in the Largest Acknowledged field, was received by this peer to when this ACK was sent. The value of the ACK Delay field is scaled by multiplying the encoded value by the 2 to the power of the value of the "ack_delay_exponent" transport parameter set by the sender of the ACK frame. The "ack_delay_exponent" defaults to 3, or a multiplier of 8 (see Section 6.4.1). Scaling in this fashion allows for a larger range of values with a shorter encoding at the cost of lower resolution.

ACK Block Count: The number of Additional ACK Block (and Gap) fields after the First ACK Block.

ACK Blocks: Contains one or more blocks of packet numbers which have been successfully received, see Section 7.15.1.

### 7.15.1. ACK Block Section

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

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

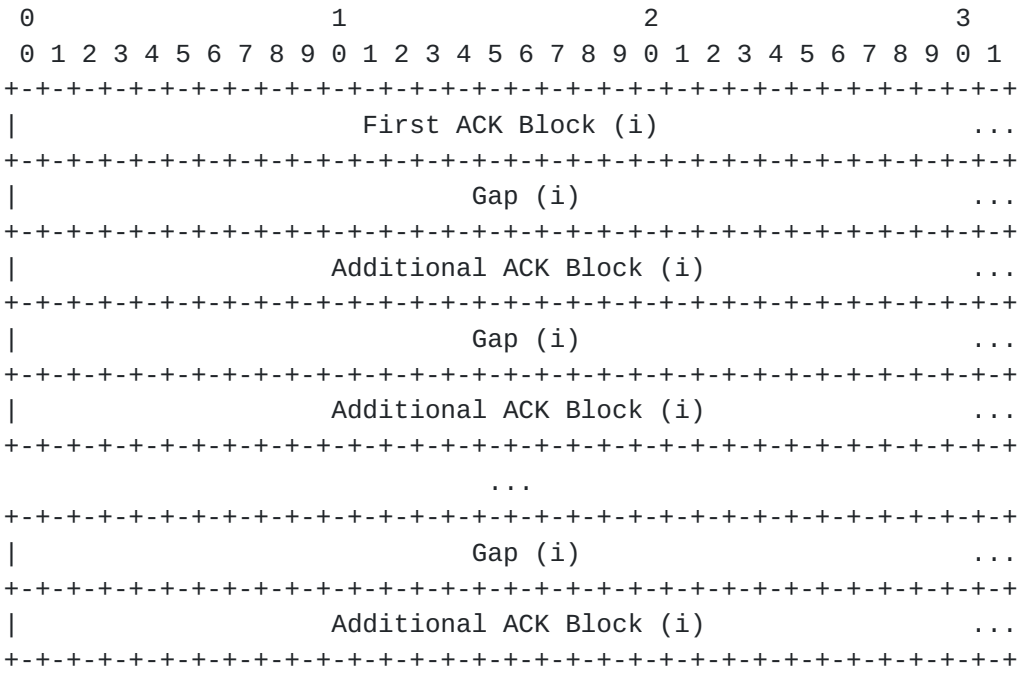



Figure 8: ACK Block Section

Each ACK Block acknowledges a contiguous range of packets by indicating the number of acknowledged packets that precede the largest packet number in that block. A value of zero indicates that only the largest packet number is acknowledged. Larger ACK Block values indicate a larger range, with corresponding lower values for the smallest packet number in the range. Thus, given a largest packet number for the ACK, the smallest value is determined by the formula:

```
smallest = largest - ack_block
```

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

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

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

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

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

If the calculated value for largest or smallest packet number for any ACK Block is negative, an endpoint MUST generate a connection error of type FRAME_ERROR indicating an error in an ACK frame (that is, 0x10d).

The fields in the ACK Block Section are:

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

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

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

### 7.15.2. Sending ACK Frames

Implementations MUST NOT generate packets that only contain ACK frames in response to packets which only contain ACK frames. However, they MUST acknowledge packets containing only ACK frames when sending ACK frames in response to other packets. Implementations MUST NOT send more than one packet containing only ACK frames per received packet that contains frames other than ACK frames. Packets containing non-ACK frames MUST be acknowledged immediately or when a delayed ack timer expires.

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

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

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

### 7.15.3. 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,

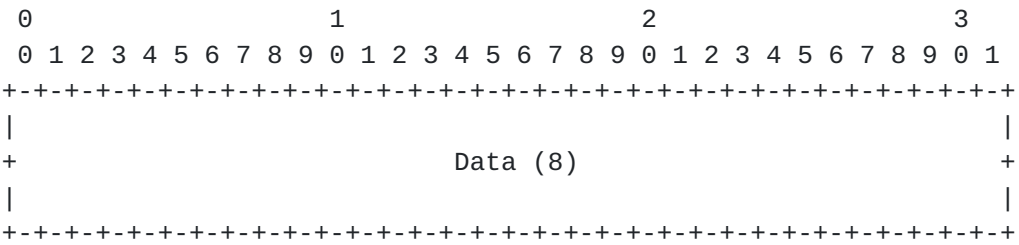
unless it is able to acknowledge those packets in later packets protected by 1-RTT keys. At the completion of the cryptographic handshake, both peers send unprotected packets containing cryptographic handshake messages followed by packets protected by 1-RTT keys. An endpoint SHOULD acknowledge the unprotected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

## 7.16. PATH_CHALLENGE Frame

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

PATH_CHALLENGE frames contain an 8-byte payload.




Data: This 8-byte field contains arbitrary data.

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

The recipient of this frame MUST generate a PATH_RESPONSE frame (Section 7.17) containing the same Data.

## 7.17. PATH_RESPONSE Frame

The PATH_RESPONSE frame (type=0x0f) is sent in response to a PATH_CHALLENGE frame. Its format is identical to the PATH_CHALLENGE frame (Section 7.16).

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

### 7.18. STREAM Frames

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

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

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

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

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

A STREAM frame is shown below.

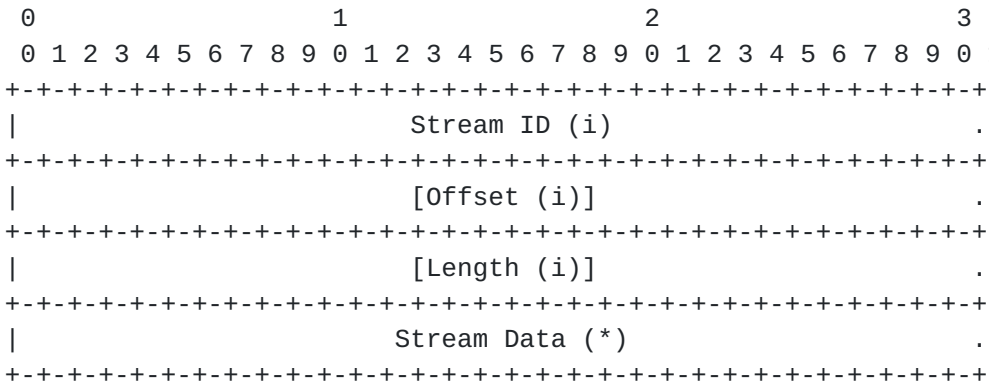



Figure 9: STREAM Frame Format

The STREAM frame contains the following fields:

Stream ID: A variable-length integer indicating the stream ID of the stream (see Section 9.1).

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

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

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

A stream frame's Stream Data MUST NOT be empty, unless the offset is 0 or 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.

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

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.

## 8. Packetization and Reliability

A sender bundles one or more frames in a QUIC packet (see Section 5).

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 knowledge about application sending behavior or heuristics to

determine whether and for how long to wait. This waiting period is an implementation decision, and an implementation should be careful to delay conservatively, since any delay is likely to increase application-visible latency.

## 8.1. Packet Processing and Acknowledgment

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

Once the packet has been fully processed, a receiver acknowledges receipt by sending one or more ACK frames containing the packet number of the received packet. To avoid creating an indefinite feedback loop, an endpoint MUST NOT send an ACK frame in response to a packet containing only ACK or PADDING frames, even if there are packet gaps which precede the received packet. The endpoint MUST acknowledge packets containing only ACK or PADDING frames in the next ACK frame that it sends.

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

## 8.2. Retransmission of Information

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

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

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

- The most recent set of acknowledgments are sent in ACK frames. An ACK frame SHOULD contain all unacknowledged acknowledgments, as described in Section 7.15.2.

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

- Similarly, a request to cancel stream transmission, as encoded in a STOP_SENDING frame, is sent until the receive stream enters either a "Data Recvd" or "Reset Recvd" state, see Section 9.3.

- Connection close signals, including those that use CONNECTION_CLOSE and APPLICATION_CLOSE frames, are not sent again when packet loss is detected, but as described in Section 6.9.

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

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

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

- Blocked signals are carried in BLOCKED, STREAM_BLOCKED, and STREAM_ID_BLOCKED frames. BLOCKED streams have connection scope, STREAM_BLOCKED frames have stream scope, and STREAM_ID_BLOCKED frames are scoped to a specific stream type. New frames are sent if packets containing the most recent frame for a scope is lost, but only while the endpoint is blocked on the corresponding limit. These frames always include the limit that is causing blocking at the time that they are transmitted.

- A liveness or path validation check using PATH_CHALLENGE frames is sent periodically until a matching PATH_RESPONSE frame is received or until there is no remaining need for liveness or path

validation checking. PATH_CHALLENGE frames include a different payload each time they are sent.

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

- New connection IDs are sent in NEW_CONNECTION_ID frames and retransmitted if the packet containing them is lost.

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

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

### 8.3. Packet Size

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

Clients MUST pad any Initial packet it sends to have a QUIC packet size of at least 1200 octets. Sending an Initial packet of this size ensures that the network path supports a reasonably sized packet, and helps reduce the amplitude of amplification attacks caused by server responses toward an unverified client address.

An Initial packet MAY exceed 1200 octets if the client knows that the Path Maximum Transmission Unit (PMTU) supports the size that it chooses.

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

### 8.4. Path Maximum Transmission Unit

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

All QUIC packets SHOULD be sized to fit within the estimated PMTU to avoid IP fragmentation or packet drops. To optimize bandwidth efficiency, endpoints SHOULD use Packetization Layer PMTU Discovery

([PLPMTUD]). Endpoints MAY use PMTU Discovery ([PMTUDv4], [PMTUDv6]) for detecting the PMTU, setting the PMTU appropriately, and storing the result of previous PMTU determinations.

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

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

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

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

#### 8.4.1. 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.

### 8.4.2. Special Considerations for Packetization Layer PMTU Discovery

The PADDING frame provides a useful option for PMTU probe packets that does not exist in other transports. PADDING frames generate acknowledgements, but their content need not be delivered reliably. PADDING frames may delay the delivery of application data, as they consume the congestion window. However, by definition their likely loss in a probe packet does not require delay-inducing retransmission of application data.

When implementing the algorithm in Section 7.2 of [RFC4821], the initial value of search_low SHOULD be consistent with the IPv6 minimum packet size. Paths that do not support this size cannot deliver Initial packets, and therefore are not QUIC-compliant.

Section 7.3 of [RFC4821] discusses tradeoffs between small and large increases in the size of probe packets. As QUIC probe packets need not contain application data, aggressive increases in probe size carry fewer consequences.

## 9. Streams: QUIC's Data Structuring Abstraction

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

There are two basic types of stream in QUIC. Unidirectional streams carry data in one direction only; bidirectional streams allow for data to be sent in both directions. Different stream identifiers are used to distinguish between unidirectional and bidirectional streams, as well as to create a separation between streams that are initiated by the client and server (see Section 9.1).

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

Stream offsets allow for the octets on a stream to be placed in order. An endpoint MUST be capable of delivering data received on a stream in order. Implementations MAY choose to offer the ability to

deliver data out of order. There is no means of ensuring ordering between octets on different streams.

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.

### 9.1. Stream Identifiers

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

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

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

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

The two type bits from a Stream ID therefore identify streams as summarized in Table 5.

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

Table 5: Stream ID Types

Stream ID 0 (0x0) is a client-initiated, bidirectional stream that is used for the cryptographic handshake. Stream 0 MUST NOT be used for application data.

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

Stream IDs are encoded as a variable-length integer (see Section 7.1).

## 9.2. Stream States

This section describes the two types of QUIC stream in terms of the states of their send or receive components. Two state machines are described: one for streams on which an endpoint transmits data (Section 9.2.1); another for streams from which an endpoint receives data (Section 9.2.2).

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

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

Note: These states are largely informative. This document uses stream states to describe rules for when and how different types of frames can be sent and the reactions that are expected when

different types of frames are received. Though these state machines are intended to be useful in implementing QUIC, these states aren't intended to constrain implementations. An implementation can define a different state machine as long as its behavior is consistent with an implementation that implements these states.

### 9.2.1. Send Stream States

Figure 10 shows the states for the part of a stream that sends data to a peer.

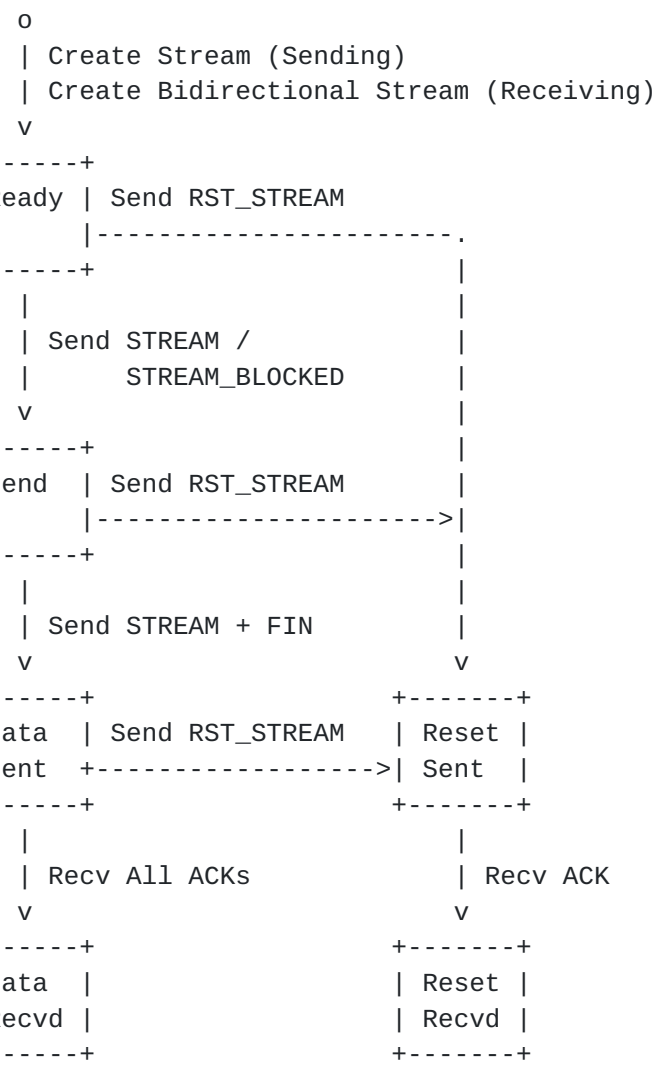



Figure 10: States for Send Streams

The sending part of stream that the endpoint initiates (types 0 and 2 for clients, 1 and 3 for servers) is opened by the application or application protocol. The "Ready" state represents a newly created

stream that is able to accept data from the application. Stream data might be buffered in this state in preparation for sending.

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

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

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

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

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

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

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

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

### 9.2.2. Receive Stream States

Figure 11 shows the states for the part of a stream that receives data from a peer. The states for a receive stream mirror only some of the states of the send stream at the peer. A receive stream doesn't track states on the send stream that cannot be observed, such as the "Ready" state; instead, receive streams track the delivery of data to the application or application protocol some of which cannot be observed by the sender.

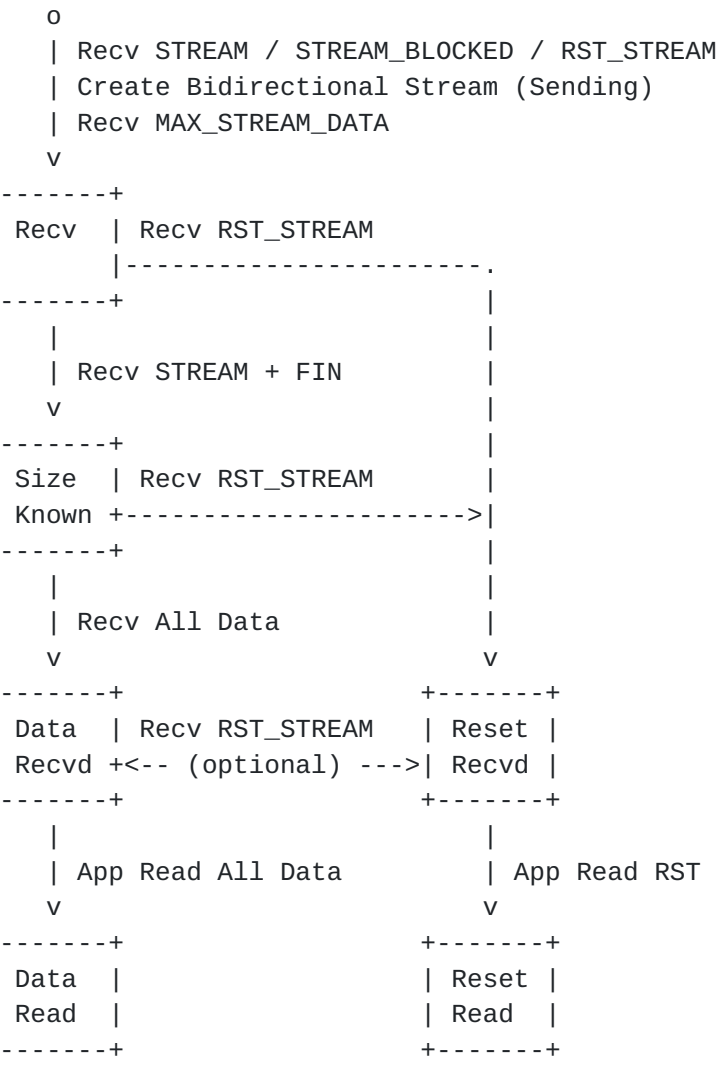



Figure 11: States for Receive Streams

The receiving part of a stream initiated by a peer (types 1 and 3 for a client, or 0 and 2 for a server) are created when the first STREAM, STREAM_BLOCKED, RST_STREAM, or MAX_STREAM_DATA (bidirectional only, see below) is received for that stream. The initial state for a

receive stream is "Recv". Receiving a RST_STREAM frame causes the receive stream to immediately transition to the "Reset Recvd".

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

A bidirectional stream also opens when a MAX_STREAM_DATA frame is received. Receiving a MAX_STREAM_DATA frame implies that the remote peer has opened the stream and is providing flow control credit. A MAX_STREAM_DATA frame might arrive before a STREAM or STREAM_BLOCKED frame if packets are lost or reordered.

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

When a STREAM frame with a FIN bit is received, the final offset (see Section 10.3) is known. The receive stream enters the "Size Known" state. In this state, the endpoint no longer needs to send MAX_STREAM_DATA frames, it only receives any retransmissions of stream data.

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

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

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

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

of stream data, discard any data that was not consumed, and signal the existence of the RST_STREAM immediately. Alternatively, the RST_STREAM signal might be suppressed or withheld if stream data is completely received. In the latter case, the receive stream effectively transitions to "Data Recvd" from "Reset Recvd".

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

### 9.2.3. Permitted Frame Types

The sender of a stream sends just three frame types that affect the state of a stream at either sender or receiver: STREAM (Section 7.18), STREAM_BLOCKED (Section 7.11), and RST_STREAM (Section 7.3).

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

The receiver of a stream sends MAX_STREAM_DATA (Section 7.7) and STOP_SENDING frames (Section 7.14).

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

### 9.2.4. Bidirectional Stream States

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

Table 6 shows a more complex mapping of bidirectional stream states that loosely correspond to the stream states in HTTP/2 [HTTP2]. This shows that multiple states on send or receive streams are mapped to the same composite state. Note that this is just one possibility for

such a mapping; this mapping requires that data is acknowledged before the transition to a "closed" or "half-closed" state.

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

Table 6: Possible Mapping of Stream States to HTTP/2

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

## 9.3. Solicited State Transitions

If an endpoint is no longer interested in the data it is receiving on a stream, it MAY send a STOP_SENDING frame identifying that stream to prompt closure of the stream in the opposite direction. This typically indicates that the receiving application is no longer

reading data it receives from the stream, but is not a guarantee that incoming data will be ignored.

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

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

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

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

## 9.4. 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 6.4.1) and is subsequently increased by MAX_STREAM_ID frames (see Section 7.8).

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 11), unless this is a result of a change in the initial offsets (see Section 6.4.2).

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.

### 9.5. 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 octet of data on a stream has an offset of 0. An endpoint is expected to send every stream octet. The largest offset delivered on a stream MUST be less than 2^62. 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. Data on stream 0 other than the initial cryptographic handshake message is still subject to stream-level data limits and MAX_STREAM_DATA. This message is exempt from flow control because it needs to be sent in a single packet regardless of the server's flow control state. This rule applies even for 0-RTT handshakes where the remembered value of MAX_STREAM_DATA would not permit sending a full initial cryptographic handshake message.

Flow control is described in detail in Section 10, and congestion control is described in the companion document [QUIC-RECOVERY].

### 9.6. 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 [HTTP2], shows that effective prioritization strategies have a significant positive impact on performance.

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

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

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

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

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

## 10. 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 [HTTP2]. A receiver advertises the number of octets it is prepared to receive on a given stream and for the entire connection. This leads to two levels of flow control in QUIC: (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 data receiver sends MAX_STREAM_DATA or MAX_DATA frames to the sender to advertise additional credit. MAX_STREAM_DATA frames send the the maximum absolute byte offset of a stream, while MAX_DATA sends the maximum sum of the absolute byte offsets of all streams other than stream 0.

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 11) if the peer violates the advertised connection or stream data limits.

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

A 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 except stream 0. A receiver advertises credit for a connection by sending a MAX_DATA frame. A receiver maintains a cumulative sum of bytes received on all contributing streams, which are used to check for flow control violations. A receiver might use a sum of bytes consumed on all contributing streams to determine the maximum data limit to be advertised.

### 10.1. 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.

#### 10.1.1. 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.

#### 10.1.2. 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 round-trip time estimate and the rate at which the receiving application consumes data, similar to common TCP implementations.

#### 10.1.3. Handshake Exemption

During the initial handshake, an endpoint could need to send a larger message on stream 0 than would ordinarily be permitted by the peer's initial stream flow control window. Since MAX_STREAM_DATA frames are not permitted in these early packets, the peer cannot provide additional flow control window in order to complete the handshake.

Endpoints MAY exceed the flow control limits on stream 0 prior to the completion of the cryptographic handshake. (That is, in Initial, Retry, and Handshake packets.) However, once the handshake is complete, endpoints MUST NOT send additional data beyond the peer's permitted offset. If the amount of data sent during the handshake exceeds the peer's maximum offset, the endpoint cannot send additional data on stream 0 until the peer has sent a MAX_STREAM_DATA frame indicating a larger maximum offset.

### 10.2. 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.

#### 10.2.1. 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 SHOULD 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 round trips before it expects the sender to get blocked.

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

### 10.3. Stream Final Offset

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

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

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

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

## 11. Error Handling

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

application-level errors can be isolated to a single stream (see Section 11.2).

The most appropriate error code (Section 11.3) 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 6.9.4) is not suitable for any error that can be signaled with a CONNECTION_CLOSE, APPLICATION_CLOSE, or RST_STREAM frame. A stateless reset MUST NOT be used by an endpoint that has the state necessary to send a frame on the connection.

## 11.1. Connection Errors

Errors that result in the connection being unusable, such as an obvious violation of protocol semantics or corruption of state that affects an entire connection, MUST be signaled using a CONNECTION_CLOSE or APPLICATION_CLOSE frame (Section 7.4, Section 7.5). An endpoint MAY close the connection in this manner even if the error only affects a single stream.

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

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

An endpoint that chooses not to retransmit packets containing CONNECTION_CLOSE or APPLICATION_CLOSE risks a peer missing the first such packet. The only mechanism available to an endpoint that continues to receive data for a terminated connection is to use the stateless reset process (Section 6.9.4).

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

### 11.2. Stream Errors

If an application-level error affects a single stream, but otherwise leaves the connection in a recoverable state, the endpoint can send a RST_STREAM frame (Section 7.3) 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.

Other than STOPPING (Section 9.3), RST_STREAM MUST be instigated by the application and MUST carry an application error code. Resetting a stream without knowledge of the application protocol could cause the protocol to enter an unrecoverable state. Application protocols might require certain streams to be reliably delivered in order to guarantee consistent state between endpoints.

### 11.3. Transport Error Codes

QUIC error codes are 16-bit unsigned integers.

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

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

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

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

FLOW_CONTROL_ERROR (0x3): An endpoint received more data than it permitted in its advertised data limits (see Section 10).

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

STREAM_STATE_ERROR (0x5): An endpoint received a frame for a stream that was not in a state that permitted that frame (see Section 9.2).

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

FRAME_FORMAT_ERROR (0x7): 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 (0x1XX) if possible.

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

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

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

UNSOLICITED_PATH_RESPONSE (0xB): An endpoint received a PATH_RESPONSE frame that did not correspond to any PATH_CHALLENGE frame that it previously sent.

FRAME_ERROR (0x1XX): 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 (0x106).

Codes for errors occuring when TLS is used for the crypto handshake are defined in Section 11 of [QUIC-TLS]. See Section 13.2 for details of registering new error codes.

### 11.4. Application Protocol Error Codes

Application protocol error codes are 16-bit unsigned integers, but the management of application error codes are left to application protocols. Application protocol error codes are used for the RST_STREAM (Section 7.3) and APPLICATION_CLOSE (Section 7.5) frames.

There is no restriction on the use of the 16-bit error code space for application protocols. However, QUIC reserves the error code with a value of 0 to mean STOPPING. The application error code of STOPPING (0) is used by the transport to cancel a stream in response to receipt of a STOP_SENDING frame.

# 12. Security and Privacy Considerations

## 12.1. Spoofed ACK Attack

An attacker might be able to receive an address validation token (Section 6.6) from the server and then release the IP address it used to acquire that token. The attacker may, in the future, spoof this same address (which now presumably addresses a different endpoint), and initiate a 0-RTT connection with a server on the victim's behalf. The attacker can then spoof ACK frames to the server which cause the server to send excessive amounts of data toward the new owner of the IP address.

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

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

## 12.2. Optimistic ACK Attack

An endpoint that acknowledges packets it has not received might cause a congestion controller to permit sending at rates beyond what the network supports. An endpoint MAY skip packet numbers when sending packets to detect this behavior. An endpoint can then immediately close the connection with a connection error of type PROTOCOL_VIOLATION (see Section 6.9.3).

### 12.3. 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.

### 12.4. 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.

### 12.5. 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 9.1. 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 9.4. 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.

# 13. IANA Considerations

## 13.1. QUIC Transport Parameter Registry

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

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

Registrations MUST include the following fields:

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

Parameter Name: A short mnemonic for the parameter.

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

The nominated expert(s) verify that a specification exists and is readily accessible. 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 7.

| Value | Parameter Name | Specification |
|---|---|---|
| 0x0000 | initial_max_stream_data | Section 6.4.1 |
| 0x0001 | initial_max_data | Section 6.4.1 |
| 0x0002 | initial_max_stream_id_bidi | Section 6.4.1 |
| 0x0003 | idle_timeout | Section 6.4.1 |
| 0x0005 | max_packet_size | Section 6.4.1 |
| 0x0006 | stateless_reset_token | Section 6.4.1 |
| 0x0007 | ack_delay_exponent | Section 6.4.1 |
| 0x0008 | initial_max_stream_id_uni | Section 6.4.1 |

Table 7: Initial QUIC Transport Parameters Entries

## 13.2. QUIC Transport Error Codes Registry

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

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

Registrations MUST include the following fields:

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

Code: A short mnemonic for the parameter.

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

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

The initial contents of this registry are shown in Table 8. Note that FRAME_ERROR takes the range from 0x100 to 0x1FF and private use occupies the range from 0xFE00 to 0xFFFF.

| Value | Error | Description | Specification |
|---|---|---|---|
| 0x0 | NO_ERROR | No error | Section 11.3 |
| 0x1 | INTERNAL_ERROR | Implementation error | Section 11.3 |
| 0x2 | SERVER_BUSY | Server currently busy | Section 11.3 |
| 0x3 | FLOW_CONTROL_ERROR | Flow control error | Section 11.3 |
| 0x4 | STREAM_ID_ERROR | Invalid stream ID | Section 11.3 |
| 0x5 | STREAM_STATE_ERROR | Frame received in invalid stream state | Section 11.3 |
| 0x6 | FINAL_OFFSET_ERROR | Change to final stream offset | Section 11.3 |
| 0x7 | FRAME_FORMAT_ERROR | Generic frame format error | Section 11.3 |
| 0x8 | TRANSPORT_PARAMETER_ERROR | Error in transport parameters | Section 11.3 |
| 0x9 | VERSION_NEGOTIATION_ERROR | Version negotiation failure | Section 11.3 |
| 0xA | PROTOCOL_VIOLATION | Generic protocol violation | Section 11.3 |
| 0xB | UNSOLICITED_PATH_RESPO | Unsolicited | Section 11.3 |

| | | | | |
|---|---|---|---|---|
| | NSE | PATH_RESPONSE frame | | |
| 0x100-0x1FF | FRAME_ERROR | Specific frame format error | Section 11.3 | |

Table 8: Initial QUIC Transport Error Codes Entries

# 14. References

## 14.1. 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.

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

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

[PMTUDv6] McCann, J., Deering, S., Mogul, J., and R. Hinden, Ed., "Path MTU Discovery for IP version 6", STD 87, RFC 8201, DOI 10.17487/RFC8201, July 2017, <https://www.rfc-editor.org/info/rfc8201>.

[QUIC-RECOVERY]
Iyengar, J., Ed. and I. Swett, Ed., "QUIC Loss Detection and Congestion Control", draft-ietf-quic-recovery-10 (work in progress), April 2018.

[QUIC-TLS]
Thomson, M., Ed. and S. Turner, Ed., "Using Transport Layer Security (TLS) to Secure QUIC", draft-ietf-quic-tls-10 (work in progress), April 2018.

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

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

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <https://www.rfc-editor.org/info/rfc3629>.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <https://www.rfc-editor.org/info/rfc4086>.

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

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

### 14.2. Informative References

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

[HTTP2] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <https://www.rfc-editor.org/info/rfc7540>.

[QUIC-INVARIANTS] Thomson, M., "Version-Independent Properties of QUIC", draft-ietf-quic-invariants-01 (work in progress), April 2018.

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

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

[RFC4787] Audet, F., Ed. and C. Jennings, "Network Address Translation (NAT) Behavioral Requirements for Unicast UDP", BCP 127, RFC 4787, DOI 10.17487/RFC4787, January 2007, <https://www.rfc-editor.org/info/rfc4787>.

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

[RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <https://www.rfc-editor.org/info/rfc7301>.

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

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

### 14.3. URIs

[1] https://mailarchive.ietf.org/arch/search/?email_list=quic

[2] https://github.com/quicwg

[3] https://github.com/quicwg/base-drafts/labels/-transport

[4] 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-10

- Swap payload length and packed number fields in long header (#1294)
- Clarified that CONNECTION_CLOSE is allowed in Handshake packet (#1274)
- Spin bit reserved (#1283)
- Coalescing multiple QUIC packets in a UDP datagram (#1262, #1285)
- A more complete connection migration (#1249)
- Refine opportunistic ACK defense text (#305, #1030, #1185)
- A Stateless Reset Token isn't mandatory (#818, #1191)
- Removed implicit stream opening (#896, #1193)
- An empty STREAM frame can be used to open a stream without sending data (#901, #1194)
- Define stream counts in transport parameters rather than a maximum stream ID (#1023, #1065)
- STOP_SENDING is now prohibited before streams are used (#1050)

- Recommend including ACK in Retry packets and allow PADDING (#1067, #882)

- Endpoints now become closing after an idle timeout (#1178, #1179)

- Remove implication that Version Negotiation is sent when a packet of the wrong version is received (#1197)

### C.2. Since draft-ietf-quic-transport-09

- Added PATH_CHALLENGE and PATH_RESPONSE frames to replace PING with Data and PONG frame. Changed ACK frame type from 0x0e to 0x0d. (#1091, #725, #1086)

- A server can now only send 3 packets without validating the client address (#38, #1090)

- Delivery order of stream data is no longer strongly specified (#252, #1070)

- Rework of packet handling and version negotiation (#1038)

- Stream 0 is now exempt from flow control until the handshake completes (#1074, #725, #825, #1082)

- Improved retransmission rules for all frame types: information is retransmitted, not packets or frames (#463, #765, #1095, #1053)

- Added an error code for server busy signals (#1137)

- Endpoints now set the connection ID that their peer uses. Connection IDs are variable length. Removed the omit_connection_id transport parameter and the corresponding short header flag. (#1089, #1052, #1146, #821, #745, #821, #1166, #1151)

### C.3. Since draft-ietf-quic-transport-08

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

- BLOCKED frame now includes reason for blocking (#452, #924, #927, #928)

- GAP limitation in ACK Frame (#613)

- Improved PMTUD description (#614, #1036)

- Clarified stream state machine (#634, #662, #743, #894)

- Reserved versions don't need to be generated deterministically (#831, #931)
- You don't always need the draining period (#871)
- Stateless reset clarified as version-specific (#930, #986)
- initial_max_stream_id_x transport parameters are optional (#970, #971)
- Ack Delay assumes a default value during the handshake (#1007, #1009)
- Removed transport parameters from NewSessionTicket (#1015)

### C.4. Since draft-ietf-quic-transport-07

- The long header now has version before packet number (#926, #939)
- Rename and consolidate packet types (#846, #822, #847)
- Packet types are assigned new codepoints and the Connection ID Flag is inverted (#426, #956)
- Removed type for Version Negotiation and use Version 0 (#963, #968)
- Streams are split into unidirectional and bidirectional (#643, #656, #720, #872, #175, #885)
  - Stream limits now have separate uni- and bi-directinal transport parameters (#909, #958)
  - Stream limit transport parameters are now optional and default to 0 (#970, #971)
- The stream state machine has been split into read and write (#634, #894)
- Employ variable-length integer encodings throughout (#595)
- Improvements to connection close
  - Added distinct closing and draining states (#899, #871)
  - Draining period can terminate early (#869, #870)
  - Clarifications about stateless reset (#889, #890)

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

### C.5. Since draft-ietf-quic-transport-06

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

### C.6. Since draft-ietf-quic-transport-05

- Stateless token is server-only (#726)
- Refactor section on connection termination (#733, #748, #328, #177)
- Limit size of Version Negotiation packet (#585)
- Clarify when and what to ack (#736)
- Renamed STREAM_ID_NEEDED to STREAM_ID_BLOCKED
- Clarify Keep-alive requirements (#729)

### C.7. 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.8. Since draft-ietf-quic-transport-03

- Change STREAM and RST_STREAM layout
- Add MAX_STREAM_ID settings

### C.9. 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.10. 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.11. 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.12. 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)
Fastly

Email: jri.ietf@gmail.com

Martin Thomson (editor)
Mozilla

Email: martin.thomson@gmail.com