QUIC: A UDP-Based Multiplexed and Secure Transport
draft-ietf-quic-transport-32

Abstract

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

Note to Readers

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

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

Status of This Memo

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

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

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

   This Internet-Draft will expire on 23 April 2021.

Copyright Notice

Table of Contents

1. Overview

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

   * Stream multiplexing

   * Stream- and connection-level flow control

   * Low-latency connection establishment

   * Connection migration and resilience to NAT rebinding

   * Authenticated and encrypted header and payload

   QUIC establishes a connection, which is a stateful interaction
   between a client and server. The primary purpose of a connection is
   to support the structured exchange of data by an application
   protocol.

   Application protocols exchange information over a QUIC connection via
   streams, which are ordered sequences of bytes. Two types of stream
   can be created: bidirectional streams, which allow both endpoints to
   send data; and unidirectional streams, which allow a single endpoint
   to send data. A credit-based scheme is used to limit stream creation
   and to bound the amount of data that can be sent.

   The QUIC handshake combines negotiation of cryptographic and
   transport parameters. The handshake is structured to permit the
   exchange of application data as soon as possible. This includes an
   option for clients to send data immediately (0-RTT), which might
   require prior communication to enable.

   QUIC connections are not strictly bound to a single network path.
   Connection migration uses connection identifiers to allow connections
   to transfer to a new network path.

   Frames are used in QUIC to communicate between endpoints. One or
   more frames are assembled into a QUIC packet. QUIC authenticates all
   packets and encrypts as much as is practical. QUIC packets are
   carried in UDP datagrams ([UDP]) to better facilitate deployment in
   existing systems and networks.

   Once established, multiple options are provided for connection
   termination. Applications can manage a graceful shutdown, endpoints
   can negotiate a timeout period, errors can cause immediate connection
   teardown, and a stateless mechanism provides for termination of
   connections after one endpoint has lost state.

1.1. Document Structure

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

   * Streams are the basic service abstraction that QUIC provides.

      - Section 2 describes core concepts related to streams,

      - Section 3 provides a reference model for stream states, and

      - Section 4 outlines the operation of flow control.

   * Connections are the context in which QUIC endpoints communicate.

      - Section 5 describes core concepts related to connections,

      - Section 6 describes version negotiation,

      - Section 7 details the process for establishing connections,

      - Section 8 describes address validation and critical denial of
         service mitigations,

      - Section 9 describes how endpoints migrate a connection to a new
         network path,

      - Section 10 lists the options for terminating an open
         connection, and

      - Section 11 provides guidance for stream and connection error
         handling.

   * Packets and frames are the basic unit used by QUIC to communicate.

      - Section 12 describes concepts related to packets and frames,

      - Section 13 defines models for the transmission, retransmission,
         and acknowledgement of data, and

      - Section 14 specifies rules for managing the size of datagrams
         carrying QUIC packets.

   * Finally, encoding details of QUIC protocol elements are described
      in:

      - Section 15 (Versions),

- Section 16 (Integer Encoding),

- Section 17 (Packet Headers),

- Section 18 (Transport Parameters),

- Section 19 (Frames), and

- Section 20 (Errors).

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

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

To refer to QUIC version 1, cite this document. References to the limited set of version-independent properties of QUIC can cite [QUIC-INVARIANTS].

1.2. Terms 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.

Commonly used terms in the document are described below.

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

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

Ack-eliciting Packet: A QUIC packet that contains frames other than ACK, PADDING, and CONNECTION_CLOSE. These cause a recipient to send an acknowledgment; see Section 13.2.1.

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

Client: The endpoint that initiates a QUIC connection.

   Server: The endpoint that accepts a QUIC connection.

   Address: When used without qualification, the tuple of IP version,
      IP address, and UDP port number that represents one end of a
      network path.

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

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

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

1.3. Notational Conventions

   Packet and frame diagrams in this document use a custom format. The
   purpose of this format is to summarize, not define, protocol
   elements. Prose defines the complete semantics and details of
   structures.

   Complex fields are named and then followed by a list of fields
   surrounded by a pair of matching braces. Each field in this list is
   separated by commas.

   Individual fields include length information, plus indications about
   fixed value, optionality, or repetitions. Individual fields use the
   following notational conventions, with all lengths in bits:

   x (A): Indicates that x is A bits long

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

   x (A..B): Indicates that x can be any length from A to B; A can be
      omitted to indicate a minimum of zero bits and B can be omitted to
      indicate no set upper limit; values in this format always end on
      an octet boundary

   x (?) = C: Indicates that x has a fixed value of C with the length
      described by ?, as above

   x (?) = C..D: Indicates that x has a value in the range from C to D,
      inclusive, with the length described by ?, as above

[x (E)]: Indicates that x is optional (and has length of E)

x (E) ...: Indicates that x is repeated zero or more times (and that
   each instance is length E)

This document uses network byte order (that is, big endian) values.
Fields are placed starting from the high-order bits of each byte.

By convention, individual fields reference a complex field by using
the name of the complex field.

For example:

```
Example Structure {
  One-bit Field (1),
  7-bit Field with Fixed Value (7) = 61,
  Field with Variable-Length Integer (i),
  Arbitrary-Length Field (..),
  Variable-Length Field (8..24),
  Field With Minimum Length (16..),
  Field With Maximum Length (..128),
  [Optional Field (64)],
  Repeated Field (8) ...,
}
```

Figure 1: Example Format

2. Streams

   Streams in QUIC provide a lightweight, ordered byte-stream
   abstraction to an application. Streams can be unidirectional or
   bidirectional.

   Streams can be created by sending data. Other processes associated
   with stream management - ending, cancelling, and managing flow
   control - are all designed to impose minimal overheads. For
   instance, a single STREAM frame (Section 19.8) can open, carry data
   for, and close a stream. Streams can also be long-lived and can last
   the entire duration of a connection.

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

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

2.1. Stream Types and Identifiers

   Streams can be unidirectional or bidirectional. Unidirectional
   streams carry data in one direction: from the initiator of the stream
   to its peer. Bidirectional streams allow for data to be sent in both
   directions.

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

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

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

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

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

                      Table 1: Stream ID Types

The stream space for each type begins at the minimum value (0x0 through 0x3 respectively); successive streams of each type are created with numerically increasing stream IDs. A stream ID that is used out of order results in all streams of that type with lower-numbered stream IDs also being opened.

2.2. Sending and Receiving Data

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

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

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

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

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

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

2.3. Stream Prioritization

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

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

A QUIC implementation SHOULD provide ways in which an application can indicate the relative priority of streams. An implementation uses information provided by the application to determine how to allocate resources to active streams.

2.4. Operations on Streams

This document does not define an API for QUIC, but instead defines a set of functions on streams that application protocols can rely upon. An application protocol can assume that a QUIC implementation provides an interface that includes the operations described in this section. An implementation designed for use with a specific application protocol might provide only those operations that are used by that protocol.

On the sending part of a stream, an application protocol can:

* write data, understanding when stream flow control credit
   (Section 4.1) has successfully been reserved to send the written
   data;

* end the stream (clean termination), resulting in a STREAM frame
   (Section 19.8) with the FIN bit set; and

* reset the stream (abrupt termination), resulting in a RESET_STREAM
   frame (Section 19.4) if the stream was not already in a terminal
   state.

On the receiving part of a stream, an application protocol can:

* read data; and

* abort reading of the stream and request closure, possibly
   resulting in a STOP_SENDING frame (Section 19.5).

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

3. Stream States

This section describes streams in terms of their send or receive components. Two state machines are described: one for the streams on which an endpoint transmits data (Section 3.1), and another for streams on which an endpoint receives data (Section 3.2).

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

The state machines shown in this section 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 are not 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.

Note: In some cases, a single event or action can cause a transition
   through multiple states. For instance, sending STREAM with a FIN
   bit set can cause two state transitions for a sending stream: from
   the Ready state to the Send state, and from the Send state to the
   Data Sent state.

3.1. Sending Stream States

Figure 2 shows the states for the part of a stream that sends data to
a peer.

```
            o
            | Create Stream (Sending)
            | Peer Creates Bidirectional Stream
            v
      +-------+
      | Ready | Send RESET_STREAM
      |       |----------------------.
      +-------+                      |
          |                         |
          | Send STREAM /           |
          |       STREAM_DATA_BLOCKED |
          |                         |
          | Peer Creates            |
          |       Bidirectional Stream |
          v                         |
      +-------+                      |
      | Send  | Send RESET_STREAM    |
      |       |---------------------->|
      +-------+                      |
          |                         |
          | Send STREAM + FIN       |
          v                         v
      +-------+              +-------+
      | Data  | Send RESET_STREAM | Reset |
      | Sent  |----------------->| Sent  |
      +-------+              +-------+
          |                      |
          | Recv All ACKs        | Recv ACK
          v                      v
      +-------+              +-------+
      | Data  |              | Reset |
      | Recvd |              | Recvd |
      +-------+              +-------+
```
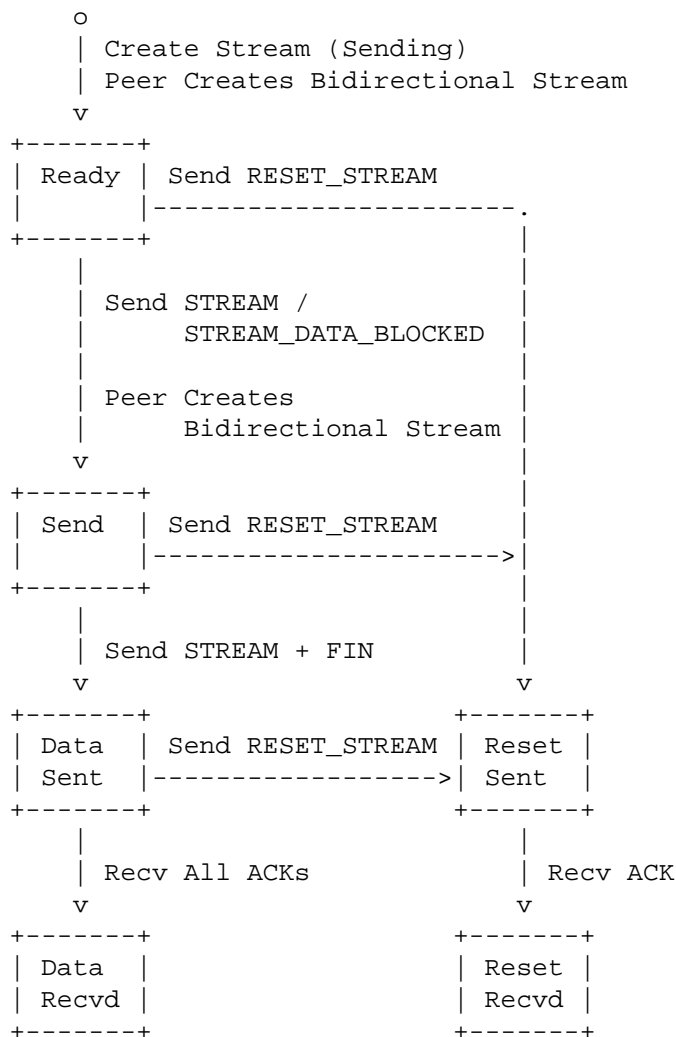
                 Figure 2: States for Sending Parts of Streams

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

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

The sending part of a bidirectional stream initiated by a peer (type 0 for a server, type 1 for a client) starts in the "Ready" state when the receiving part is created.

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

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

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

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

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

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

3.2. Receiving Stream States

   Figure 3 shows the states for the part of a stream that receives data
   from a peer. The states for a receiving part of a stream mirror only
   some of the states of the sending part of the stream at the peer.
   The receiving part of a stream does not track states on the sending
   part that cannot be observed, such as the "Ready" state. Instead,
   the receiving part of a stream tracks the delivery of data to the
   application, some of which cannot be observed by the sender.

```
            o
            | Recv STREAM / STREAM_DATA_BLOCKED / RESET_STREAM
            | Create Bidirectional Stream (Sending)
            | Recv MAX_STREAM_DATA / STOP_SENDING (Bidirectional)
            | Create Higher-Numbered Stream
            v
       +-------+
       | Recv  | Recv RESET_STREAM
       |       |-----------------------.
       +-------+                       |
           |                           |
           | Recv STREAM + FIN         |
           v                           |
       +-------+                       |
       | Size  | Recv RESET_STREAM     |
       | Known |---------------------->|
       +-------+                       |
           |                           |
           | Recv All Data             |
           v                           v
       +-------+ Recv RESET_STREAM +-------+
       | Data  |--- (optional) --->| Reset |
       | Recvd |   Recv All Data   | Recvd |
       +-------+<-- (optional) ----+-------+
           |                           |
           | App Read All Data         | App Read RST
           v                           v
       +-------+                   +-------+
       | Data  |                   | Reset |
       | Read  |                   | Read  |
       +-------+                   +-------+
```

              Figure 3: States for Receiving Parts of Streams

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

MAX_STREAM_DATA or STOP_SENDING frame for the sending part of the
stream also creates the receiving part. The initial state for the
receiving part of a stream is "Recv".

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

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

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

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

When a STREAM frame with a FIN bit is received, the final size of the
stream is known; see Section 4.5. The receiving part of the stream
then enters the "Size Known" state. In this state, the endpoint no
longer needs to send MAX_STREAM_DATA frames, it only receives any
retransmissions of stream data.

Once all data for the stream has been received, the receiving part
enters the "Data Recvd" state. This might happen as a result of
receiving the same STREAM frame that causes the transition to "Size
Known". After all data has been received, any STREAM or
STREAM_DATA_BLOCKED frames for the stream can be discarded.

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

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

It is possible that all stream data has already been received when a
RESET_STREAM is received (that is, in the "Data Recvd" state).
Similarly, it is possible for remaining stream data to arrive after
receiving a RESET_STREAM frame (the "Reset Recvd" state). An
implementation is free to manage this situation as it chooses.

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

Once the application receives the signal indicating that the stream
was reset, the receiving part of the stream transitions to the "Reset
Read" state, which is a terminal state.

3.3. Permitted Frame Types

The sender of a stream sends just three frame types that affect the
state of a stream at either sender or receiver: STREAM
(Section 19.8), STREAM_DATA_BLOCKED (Section 19.13), and RESET_STREAM
(Section 19.4).

A sender MUST NOT send any of these frames from a terminal state
("Data Recvd" or "Reset Recvd"). A sender MUST NOT send a STREAM or
STREAM_DATA_BLOCKED frame for a stream in the "Reset Sent" state or
any terminal state, that is, after sending a RESET_STREAM frame. A
receiver could receive any of these three frames in any state, due to
the possibility of delayed delivery of packets carrying them.

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

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

3.4. Bidirectional Stream States

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

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

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

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

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

3.5. Solicited State Transitions

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

If the stream is in the "Recv" or "Size Known" states, the transport
SHOULD signal this by sending a STOP_SENDING frame to prompt closure
of the stream in the opposite direction. This typically indicates
that the receiving application is no longer reading data it receives
from the stream, but it is not a guarantee that incoming data will be
ignored.

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

A STOP_SENDING frame requests that the receiving endpoint send a
RESET_STREAM frame. An endpoint that receives a STOP_SENDING frame
MUST send a RESET_STREAM frame if the stream is in the Ready or Send
state. If the stream is in the "Data Sent" state, the endpoint MAY
defer sending the RESET_STREAM frame until the packets containing
outstanding data are acknowledged or declared lost. If any
outstanding data is declared lost, the endpoint SHOULD send a
RESET_STREAM frame instead of retransmitting the data.

An endpoint SHOULD copy the error code from the STOP_SENDING frame to
the RESET_STREAM frame it sends, but MAY use any application error
code. An endpoint that sends a STOP_SENDING frame MAY ignore the
error code in any RESET_STREAM frames subsequently received for that
stream.

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

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

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

4. Flow Control

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

   Similarly, to limit concurrency within a connection, a QUIC endpoint
   controls the maximum cumulative number of streams that its peer can
   initiate, as described in Section 4.6.

   Data sent in CRYPTO frames is not flow controlled in the same way as
   stream data. QUIC relies on the cryptographic protocol
   implementation to avoid excessive buffering of data; see [QUIC-TLS].
   To avoid excessive buffering at multiple layers, QUIC implementations
   SHOULD provide an interface for the cryptographic protocol
   implementation to communicate its buffering limits.

4.1. Data Flow Control

   QUIC employs a limit-based flow-control scheme where a receiver
   advertises the limit of total bytes it is prepared to receive on a
   given stream or for the entire connection. This leads to two levels
   of data flow control in QUIC:

   * Stream flow control, which prevents a single stream from consuming
      the entire receive buffer for a connection by limiting the amount
      of data that can be sent on any stream.

   * Connection flow control, which prevents senders from exceeding a
      receiver's buffer capacity for the connection, by limiting the
      total bytes of stream data sent in STREAM frames on all streams.

   Senders MUST NOT send data in excess of either limit.

   A receiver sets initial limits for all streams through transport
   parameters during the handshake (Section 7.4). Subsequently, a
   receiver sends MAX_STREAM_DATA (Section 19.10) or MAX_DATA
   (Section 19.9) frames to the sender to advertise larger limits.

A receiver can advertise a larger limit for a stream by sending a
MAX_STREAM_DATA frame with the corresponding stream ID. A
MAX_STREAM_DATA frame indicates the maximum absolute byte offset of a
stream. A receiver could determine the flow control offset to be
advertised based on the current offset of data consumed on that
stream.

A receiver can advertise a larger limit for a connection by sending a
MAX_DATA frame, which indicates the maximum of the sum of the
absolute byte offsets of all streams. A receiver maintains a
cumulative sum of bytes received on all streams, which is used to
check for violations of the advertised connection or stream data
limits. A receiver could determine the maximum data limit to be
advertised based on the sum of bytes consumed on all streams.

Once a receiver advertises a limit for the connection or a stream, it
MAY advertise a smaller limit, but this has no effect.

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

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

If a sender has sent data up to the limit, it will be unable to send
new data and is considered blocked. A sender SHOULD send a
STREAM_DATA_BLOCKED or DATA_BLOCKED frame to indicate to the receiver
that it has data to write but is blocked by flow control limits. If
a sender is blocked for a period longer than the idle timeout
(Section 10.1), the receiver might close the connection even when the
sender has data that is available for transmission. To keep the
connection from closing, a sender that is flow control limited SHOULD
periodically send a STREAM_DATA_BLOCKED or DATA_BLOCKED frame when it
has no ack-eliciting packets in flight.

4.2. Increasing Flow Control Limits

Implementations decide when and how much credit to advertise in
MAX_STREAM_DATA and MAX_DATA frames, but this section offers a few
considerations.

To avoid blocking a sender, a receiver MAY send a MAX_STREAM_DATA or
MAX_DATA frame multiple times within a round trip or send it early
enough to allow time for loss of the frame and subsequent recovery.

Control frames contribute to connection overhead. Therefore,
frequently sending MAX_STREAM_DATA and MAX_DATA frames with small
changes is undesirable. On the other hand, if updates are less
frequent, larger increments to limits are necessary to avoid blocking
a sender, requiring larger resource commitments at the receiver.
There is a trade-off between resource commitment and overhead when
determining how large a limit is advertised.

A receiver can use an autotuning mechanism to tune the frequency and
amount of advertised additional credit based on a round-trip time
estimate and the rate at which the receiving application consumes
data, similar to common TCP implementations. As an optimization, an
endpoint could send frames related to flow control only when there
are other frames to send, ensuring that flow control does not cause
extra packets to be sent.

A blocked sender is not required to send STREAM_DATA_BLOCKED or
DATA_BLOCKED frames. Therefore, a receiver MUST NOT wait for a
STREAM_DATA_BLOCKED or DATA_BLOCKED frame before sending a
MAX_STREAM_DATA or MAX_DATA frame; doing so could result in the
sender being blocked for the rest of the connection. Even if the
sender sends these frames, waiting for them will result in the sender
being blocked for at least an entire round trip.

When a sender receives credit after being blocked, it might be able
to send a large amount of data in response, resulting in short-term
congestion; see Section 6.9 in [QUIC-RECOVERY] for a discussion of
how a sender can avoid this congestion.

4.3. Flow Control Performance

If an endpoint cannot ensure that its peer always has available flow
control credit that is greater than the peer's bandwidth-delay
product on this connection, its receive throughput will be limited by
flow control.

Packet loss can cause gaps in the receive buffer, preventing the
application from consuming data and freeing up receive buffer space.

Sending timely updates of flow control limits can improve
performance. Sending packets only to provide flow control updates
can increase network load and adversely affect performance. Sending
flow control updates along with other frames, such as ACK frames,
reduces the cost of those updates.

4.4. Handling Stream Cancellation

   Endpoints need to eventually agree on the amount of flow control
   credit that has been consumed on every stream, to be able to account
   for all bytes for connection-level flow control.

   On receipt of a RESET_STREAM frame, an endpoint will tear down state
   for the matching stream and ignore further data arriving on that
   stream.

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

4.5. Stream Final Size

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

   A sender always communicates the final size of a stream to the
   receiver reliably, no matter how the stream is terminated. The final
   size is the sum of the Offset and Length fields of a STREAM frame
   with a FIN flag, noting that these fields might be implicit.
   Alternatively, the Final Size field of a RESET_STREAM frame carries
   this value. This guarantees that both endpoints agree on how much
   flow control credit was consumed by the sender on that stream.

   An endpoint will know the final size for a stream when the receiving
   part of the stream enters the "Size Known" or "Reset Recvd" state
   (Section 3). The receiver MUST use the final size of the stream to
   account for all bytes sent on the stream in its connection level flow
   controller.

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

Once a final size for a stream is known, it cannot change. If a
RESET_STREAM or STREAM frame is received indicating a change in the
final size for the stream, an endpoint SHOULD respond with a
FINAL_SIZE_ERROR error; see Section 11. A receiver SHOULD treat
receipt of data at or beyond the final size as a FINAL_SIZE_ERROR
error, even after a stream is closed. Generating these errors is not
mandatory, because requiring that an endpoint generate these errors
also means that the endpoint needs to maintain the final size state
for closed streams, which could mean a significant state commitment.

4.6. Controlling Concurrency

An endpoint limits the cumulative number of incoming streams a peer
can open. Only streams with a stream ID less than (max_stream * 4 +
initial_stream_id_for_type) can be opened; see Table 1. Initial
limits are set in the transport parameters; see Section 18.2.
Subsequent limits are advertised using MAX_STREAMS frames; see
Section 19.11. Separate limits apply to unidirectional and
bidirectional streams.

If a max_streams transport parameter or a MAX_STREAMS frame is
received with a value greater than 2^60, this would allow a maximum
stream ID that cannot be expressed as a variable-length integer; see
Section 16. If either is received, the connection MUST be closed
immediately with a connection error of type TRANSPORT_PARAMETER_ERROR
if the offending value was received in a transport parameter or of
type FRAME_ENCODING_ERROR if it was received in a frame; see
Section 10.2.

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

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

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

An endpoint that is unable to open a new stream due to the peer's
limits SHOULD send a STREAMS_BLOCKED frame (Section 19.14). This
signal is considered useful for debugging. An endpoint MUST NOT wait
to receive this signal before advertising additional credit, since

doing so will mean that the peer will be blocked for at least an entire round trip, and potentially indefinitely if the peer chooses not to send STREAMS_BLOCKED frames.

5. Connections

A QUIC connection is shared state between a client and a server.

Each connection starts with a handshake phase, during which the two endpoints establish a shared secret using the cryptographic handshake protocol [QUIC-TLS] and negotiate the application protocol. The handshake (Section 7) confirms that both endpoints are willing to communicate (Section 8.1) and establishes parameters for the connection (Section 7.4).

An application protocol can use the connection during the handshake phase with some limitations. 0-RTT allows application data to be sent by a client before receiving a response from the server. However, 0-RTT provides no protection against replay attacks; see Section 9.2 of [QUIC-TLS]. A server can also send application data to a client before it receives the final cryptographic handshake messages that allow it to confirm the identity and liveness of the client. These capabilities allow an application protocol to offer the option of trading some security guarantees for reduced latency.

The use of connection IDs (Section 5.1) allows connections to migrate to a new network path, both as a direct choice of an endpoint and when forced by a change in a middlebox. Section 9 describes mitigations for the security and privacy issues associated with migration.

For connections that are no longer needed or desired, there are several ways for a client and server to terminate a connection, as described in Section 10.

5.1. Connection ID

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

The primary function of a connection ID is to ensure that changes in addressing at lower protocol layers (UDP, IP) do not cause packets for a QUIC connection to be delivered to the wrong endpoint. Each endpoint selects connection IDs using an implementation-specific (and perhaps deployment-specific) method that will allow packets with that connection ID to be routed back to the endpoint and to be identified by the endpoint upon receipt.

Connection IDs MUST NOT contain any information that can be used by an external observer (that is, one that does not cooperate with the issuer) to correlate them with other connection IDs for the same connection. As a trivial example, this means the same connection ID MUST NOT be issued more than once on the same connection.

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

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

A Version Negotiation (Section 17.2.1) packet echoes the connection IDs selected by the client, both to ensure correct routing toward the client and to demonstrate that the packet is in response to a packet sent by the client.

A zero-length connection ID can be used when a connection ID is not needed to route to the correct endpoint. However, multiplexing connections on the same local IP address and port while using zero-length connection IDs will cause failures in the presence of peer connection migration, NAT rebinding, and client port reuse. An endpoint MUST NOT use the same IP address and port for multiple connections with zero-length connection IDs, unless it is certain that those protocol features are not in use.

When an endpoint uses a non-zero-length connection ID, it needs to ensure that the peer has a supply of connection IDs from which to choose for packets sent to the endpoint. These connection IDs are supplied by the endpoint using the NEW_CONNECTION_ID frame (Section 19.15).

5.1.1. Issuing Connection IDs

   Each Connection ID has an associated sequence number to assist in
   detecting when NEW_CONNECTION_ID or RETIRE_CONNECTION_ID frames refer
   to the same value. The initial connection ID issued by an endpoint
   is sent in the Source Connection ID field of the long packet header
   (Section 17.2) during the handshake. The sequence number of the
   initial connection ID is 0. If the preferred_address transport
   parameter is sent, the sequence number of the supplied connection ID
   is 1.

   Additional connection IDs are communicated to the peer using
   NEW_CONNECTION_ID frames (Section 19.15). The sequence number on
   each newly issued connection ID MUST increase by 1. The connection
   ID randomly selected by the client in the Initial packet and any
   connection ID provided by a Retry packet are not assigned sequence
   numbers unless a server opts to retain them as its initial connection
   ID.

   When an endpoint issues a connection ID, it MUST accept packets that
   carry this connection ID for the duration of the connection or until
   its peer invalidates the connection ID via a RETIRE_CONNECTION_ID
   frame (Section 19.16). Connection IDs that are issued and not
   retired are considered active; any active connection ID is valid for
   use with the current connection at any time, in any packet type.
   This includes the connection ID issued by the server via the
   preferred_address transport parameter.

   An endpoint SHOULD ensure that its peer has a sufficient number of
   available and unused connection IDs. Endpoints advertise the number
   of active connection IDs they are willing to maintain using the
   active_connection_id_limit transport parameter. An endpoint MUST NOT
   provide more connection IDs than the peer's limit. An endpoint MAY
   send connection IDs that temporarily exceed a peer's limit if the
   NEW_CONNECTION_ID frame also requires the retirement of any excess,
   by including a sufficiently large value in the Retire Prior To field.

   A NEW_CONNECTION_ID frame might cause an endpoint to add some active
   connection IDs and retire others based on the value of the Retire
   Prior To field. After processing a NEW_CONNECTION_ID frame and
   adding and retiring active connection IDs, if the number of active
   connection IDs exceeds the value advertised in its
   active_connection_id_limit transport parameter, an endpoint MUST
   close the connection with an error of type CONNECTION_ID_LIMIT_ERROR.

   An endpoint SHOULD supply a new connection ID when the peer retires a
   connection ID. If an endpoint provided fewer connection IDs than the
   peer's active_connection_id_limit, it MAY supply a new connection ID

when it receives a packet with a previously unused connection ID. An
endpoint MAY limit the total number of connection IDs issued for each
connection to avoid the risk of running out of connection IDs; see
Section 10.3.2. An endpoint MAY also limit the issuance of
connection IDs to reduce the amount of per-path state it maintains,
such as path validation status, as its peer might interact with it
over as many paths as there are issued connection IDs.

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

5.1.2. Consuming and Retiring Connection IDs

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

An endpoint maintains a set of connection IDs received from its peer,
any of which it can use when sending packets. When the endpoint
wishes to remove a connection ID from use, it sends a
RETIRE_CONNECTION_ID frame to its peer. Sending a
RETIRE_CONNECTION_ID frame indicates that the connection ID will not
be used again and requests that the peer replace it with a new
connection ID using a NEW_CONNECTION_ID frame.

As discussed in Section 9.5, endpoints limit the use of a connection
ID to packets sent from a single local address to a single
destination address. Endpoints SHOULD retire connection IDs when
they are no longer actively using either the local or destination
address for which the connection ID was used.

An endpoint might need to stop accepting previously issued connection
IDs in certain circumstances. Such an endpoint can cause its peer to
retire connection IDs by sending a NEW_CONNECTION_ID frame with an
increased Retire Prior To field. The endpoint SHOULD continue to
accept the previously issued connection IDs until they are retired by
the peer. If the endpoint can no longer process the indicated
connection IDs, it MAY close the connection.

Upon receipt of an increased Retire Prior To field, the peer MUST
stop using the corresponding connection IDs and retire them with
RETIRE_CONNECTION_ID frames before adding the newly provided
connection ID to the set of active connection IDs. This ordering
allows an endpoint to replace all active connection IDs without the

possibility of a peer having no available connection IDs and without
exceeding the limit the peer sets in the active_connection_id_limit
transport parameter; see Section 18.2. Failure to cease using the
connection IDs when requested can result in connection failures, as
the issuing endpoint might be unable to continue using the connection
IDs with the active connection.

An endpoint SHOULD limit the number of connection IDs it has retired
locally and have not yet been acknowledged. An endpoint SHOULD allow
for sending and tracking a number of RETIRE_CONNECTION_ID frames of
at least twice the active_connection_id limit. An endpoint MUST NOT
forget a connection ID without retiring it, though it MAY choose to
treat having connection IDs in need of retirement that exceed this
limit as a connection error of type CONNECTION_ID_LIMIT_ERROR.

Endpoints SHOULD NOT issue updates of the Retire Prior To field
before receiving RETIRE_CONNECTION_ID frames that retire all
connection IDs indicated by the previous Retire Prior To value.

5.2. Matching Packets to Connections

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

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

If the Destination Connection ID is zero length and the addressing
information in the packet matches the addressing information the
endpoint uses to identify a connection with a zero-length connection
ID, QUIC processes the packet as part of that connection. An
endpoint can use just destination IP and port or both source and
destination addresses for identification, though this makes
connections fragile as described in Section 5.1.

Endpoints can send a Stateless Reset (Section 10.3) for any packets
that cannot be attributed to an existing connection. A stateless
reset allows a peer to more quickly identify when a connection
becomes unusable.

Packets that are matched to an existing connection are discarded if
the packets are inconsistent with the state of that connection. For
example, packets are discarded if they indicate a different protocol
version than that of the connection, or if the removal of packet
protection is unsuccessful once the expected keys are available.

Invalid packets that lack strong integrity protection, such as
Initial, Retry, or Version Negotiation, MAY be discarded. An
endpoint MUST generate a connection error if processing the contents
of these packets prior to discovering an error, unless it fully
reverts these changes.

5.2.1. Client Packet Handling

Valid packets sent to clients always include a Destination Connection
ID that matches a value the client selects. Clients that choose to
receive zero-length connection IDs can use the local address and port
to identify a connection. Packets that do not match an existing
connection, based on Destination Connection ID or, if this value is
zero-length, local IP address and port, are discarded.

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

If a client receives a packet that uses a different version than it
initially selected, it MUST discard that packet.

5.2.2. Server Packet Handling

If a server receives a packet that indicates an unsupported version
but is large enough to initiate a new connection for any supported
version, the server SHOULD send a Version Negotiation packet as
described in Section 6.1. A server MAY limit the number of packets
to which it responds with a Version Negotiation packet. Servers MUST
drop smaller packets that specify unsupported versions.

The first packet for an unsupported version can use different
semantics and encodings for any version-specific field. In
particular, different packet protection keys might be used for
different versions. Servers that do not support a particular version
are unlikely to be able to decrypt the payload of the packet or
properly interpret the result. Servers SHOULD respond with a Version
Negotiation packet, provided that the datagram is sufficiently long.

   Packets with a supported version, or no version field, are matched to
   a connection using the connection ID or - for packets with zero-
   length connection IDs - the local address and port. These packets
   are processed using the selected connection; otherwise, the server
   continues below.

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

   If a server refuses to accept a new connection, it SHOULD send an
   Initial packet containing a CONNECTION_CLOSE frame with error code
   CONNECTION_REFUSED.

   If the packet is a 0-RTT packet, the server MAY buffer a limited
   number of these packets in anticipation of a late-arriving Initial
   packet. Clients are not able to send Handshake packets prior to
   receiving a server response, so servers SHOULD ignore any such
   packets.

   Servers MUST drop incoming packets under all other circumstances.

5.2.3. Considerations for Simple Load Balancers

   A server deployment could load balance among servers using only
   source and destination IP addresses and ports. Changes to the
   client's IP address or port could result in packets being forwarded
   to the wrong server. Such a server deployment could use one of the
   following methods for connection continuity when a client's address
   changes.

   * Servers could use an out-of-band mechanism to forward packets to
      the correct server based on Connection ID.

   * If servers can use a dedicated server IP address or port, other
      than the one that the client initially connects to, they could use
      the preferred_address transport parameter to request that clients
      move connections to that dedicated address. Note that clients
      could choose not to use the preferred address.

   A server in a deployment that does not implement a solution to
   maintain connection continuity when the client address changes SHOULD
   indicate migration is not supported using the
   disable_active_migration transport parameter. The
   disable_active_migration transport parameter does not prohibit
   connection migration after a client has acted on a preferred_address
   transport parameter.

   Server deployments that use this simple form of load balancing MUST
   avoid the creation of a stateless reset oracle; see Section 21.10.

5.3. Operations on Connections

   This document does not define an API for QUIC, but instead defines a
   set of functions for QUIC connections that application protocols can
   rely upon. An application protocol can assume that an implementation
   of QUIC provides an interface that includes the operations described
   in this section. An implementation designed for use with a specific
   application protocol might provide only those operations that are
   used by that protocol.

   When implementing the client role, an application protocol can:

   * open a connection, which begins the exchange described in
      Section 7;

   * enable Early Data when available; and

   * be informed when Early Data has been accepted or rejected by a
      server.

   When implementing the server role, an application protocol can:

   * listen for incoming connections, which prepares for the exchange
      described in Section 7;

   * if Early Data is supported, embed application-controlled data in
      the TLS resumption ticket sent to the client; and

   * if Early Data is supported, retrieve application-controlled data
      from the client's resumption ticket and accept or reject Early
      Data based on that information.

   In either role, an application protocol can:

   * configure minimum values for the initial number of permitted
      streams of each type, as communicated in the transport parameters
      (Section 7.4);

   * control resource allocation for receive buffers by setting flow
      control limits both for streams and for the connection

   * identify whether the handshake has completed successfully or is
      still ongoing;

   * keep a connection from silently closing, either by generating PING
      frames (Section 19.2) or by requesting that the transport send
      additional frames before the idle timeout expires (Section 10.1);
      and

   * immediately close (Section 10.2) the connection.

6. Version Negotiation

   Version negotiation allows a server to indicate that it does not
   support the version the client used. A server sends a Version
   Negotiation packet in response to each packet that might initiate a
   new connection; see Section 5.2 for details.

   The size of the first packet sent by a client will determine whether
   a server sends a Version Negotiation packet. Clients that support
   multiple QUIC versions SHOULD ensure that the first UDP datagram they
   send is sized to the largest of the minimum datagram sizes from all
   versions they support, using PADDING frames (Section 19.1) as
   necessary. This ensures that the server responds if there is a
   mutually supported version. A server might not send a Version
   Negotiation packet if the datagram it receives is smaller than the
   minimum size specified in a different version; see Section 14.1.

6.1. Sending Version Negotiation Packets

   If the version selected by the client is not acceptable to the
   server, the server responds with a Version Negotiation packet; see
   Section 17.2.1. This includes a list of versions that the server
   will accept. An endpoint MUST NOT send a Version Negotiation packet
   in response to receiving a Version Negotiation packet.

   This system allows a server to process packets with unsupported
   versions without retaining state. Though either the Initial packet
   or the Version Negotiation packet that is sent in response could be
   lost, the client will send new packets until it successfully receives
   a response or it abandons the connection attempt. As a result, the
   client discards all state for the connection and does not send any
   more packets on the connection.

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

6.2. Handling Version Negotiation Packets

   Version Negotiation packets are designed to allow future versions of
   QUIC to negotiate the version in use between endpoints. Future
   versions of QUIC might change how implementations that support
   multiple versions of QUIC react to Version Negotiation packets when
   attempting to establish a connection using this version.

   A client that supports only this version of QUIC MUST abandon the
   current connection attempt if it receives a Version Negotiation
   packet, with the following two exceptions. A client MUST discard any
   Version Negotiation packet if it has received and successfully
   processed any other packet, including an earlier Version Negotiation
   packet. A client MUST discard a Version Negotiation packet that
   lists the QUIC version selected by the client.

   How to perform version negotiation is left as future work defined by
   future versions of QUIC. In particular, that future work will ensure
   robustness against version downgrade attacks; see Section 21.11.

6.2.1. Version Negotiation Between Draft Versions

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

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

   The client MUST check that the Destination and Source Connection ID
   fields match the Source and Destination Connection ID fields in a
   packet that the client sent. If this check fails, the packet MUST be
   discarded.

   Once the Version Negotiation packet is determined to be valid, the
   client then selects an acceptable protocol version from the list
   provided by the server. The client then attempts to create a new
   connection using that version. The new connection MUST use a new
   random Destination Connection ID different from the one it had
   previously sent.

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

6.3. Using Reserved Versions

   For a server to use a new version in the future, clients need to
   correctly handle unsupported versions. Some version numbers
   (0x?a?a?a?a as defined in Section 15) are reserved for inclusion in
   fields that contain version numbers.

   Endpoints MAY add reserved versions to any field where unknown or
   unsupported versions are ignored to test that a peer correctly
   ignores the value. For instance, an endpoint could include a
   reserved version in a Version Negotiation packet; see Section 17.2.1.
   Endpoints MAY send packets with a reserved version to test that a
   peer correctly discards the packet.

7. Cryptographic and Transport Handshake

   QUIC relies on a combined cryptographic and transport handshake to
   minimize connection establishment latency. QUIC uses the CRYPTO
   frame (Section 19.6) to transmit the cryptographic handshake.
   Version 0x00000001 of QUIC uses TLS as described in [QUIC-TLS]; a
   different QUIC version number could indicate that a different
   cryptographic handshake protocol is in use.

   QUIC provides reliable, ordered delivery of the cryptographic
   handshake data. QUIC packet protection is used to encrypt as much of
   the handshake protocol as possible. The cryptographic handshake MUST
   provide the following properties:

   * authenticated key exchange, where

      - a server is always authenticated,

      - a client is optionally authenticated,

      - every connection produces distinct and unrelated keys, and

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

   * authenticated values for transport parameters of both endpoints,
      and confidentiality protection for server transport parameters
      (see Section 7.4)

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

Endpoints can use packets sent during the handshake to test for
Explicit Congestion Notification (ECN) support; see Section 13.4. An
endpoint verifies support for ECN by observing whether the ACK frames
acknowledging the first packets it sends carry ECN counts, as
described in Section 13.4.2.

The CRYPTO frame can be sent in different packet number spaces
(Section 12.3). The offsets used by CRYPTO frames to ensure ordered
delivery of cryptographic handshake data start from zero in each
packet number space.

Figure 4 shows a simplified handshake and the exchange of packets and
frames that are used to advance the handshake. Exchange of
application data during the handshake is enabled where possible,
shown with a '*'. Once completed, endpoints are able to exchange
application data.

```
Client                                                   Server

Initial (CRYPTO)
0-RTT (*)                 ---------->
                                               Initial (CRYPTO)
                                             Handshake (CRYPTO)
                          <----------              1-RTT (*)
Handshake (CRYPTO)
1-RTT (*)                 ---------->
                          <---------- 1-RTT (HANDSHAKE_DONE,*)

1-RTT (*)                 <========>              1-RTT (*)
```
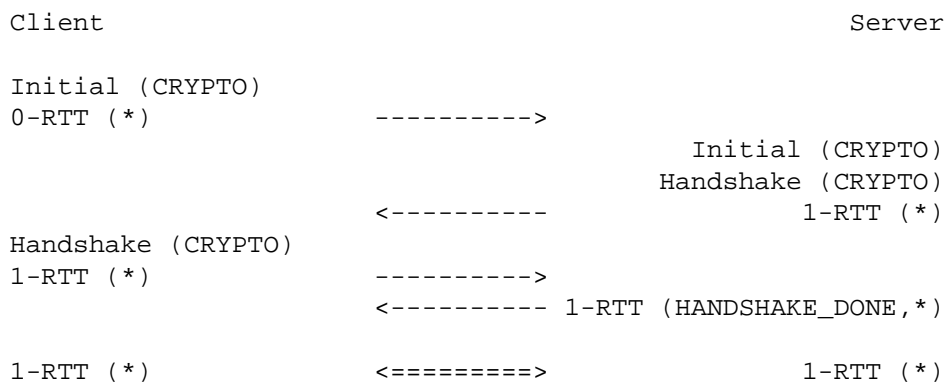
                    Figure 4: Simplified QUIC Handshake

An endpoint validates support for Explicit Congestion Notification
(ECN) by observing whether the ACK frames acknowledging the first
packets it sends carry ECN counts, as described in Section 13.4.2.

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

7.1. Example Handshake Flows

Details of how TLS is integrated with QUIC are provided in
[QUIC-TLS], but some examples are provided here. An extension of
this exchange to support client address validation is shown in
Section 8.1.2.

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

   Figure 5 provides an overview of the 1-RTT handshake. Each line
   shows a QUIC packet with the packet type and packet number shown
   first, followed by the frames that are typically contained in those
   packets. So, for instance the first packet is of type Initial, with
   packet number 0, and contains a CRYPTO frame carrying the
   ClientHello.

   Multiple QUIC packets - even of different packet types - can be
   coalesced into a single UDP datagram; see Section 12.2. As a result,
   this handshake may consist of as few as 4 UDP datagrams, or any
   number more (subject to limits inherent to the protocol, such as
   congestion control or anti-amplification). For instance, the
   server's first flight contains Initial packets, Handshake packets,
   and "0.5-RTT data" in 1-RTT packets with a short header.

   Client   Server

   Initial[0]: CRYPTO[CH] ->

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

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

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

                        Figure 5: Example 1-RTT Handshake

   Figure 6 shows an example of a connection with a 0-RTT handshake and
   a single packet of 0-RTT data. Note that as described in
   Section 12.3, the server acknowledges 0-RTT data in 1-RTT packets,
   and the client sends 1-RTT packets in the same packet number space.

```
   Client                                              Server

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

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

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

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

                       Figure 6: Example 0-RTT Handshake

7.2. Negotiating Connection IDs

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

   During the handshake, packets with the long header (Section 17.2) are
   used to establish the connection IDs used by both endpoints. 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. After processing the first Initial
   packet, each endpoint sets the Destination Connection ID field in
   subsequent packets it sends to the value of the Source Connection ID
   field that it received.

   When an Initial packet is sent by a client that has not previously
   received an Initial or Retry packet from the server, the client
   populates the Destination Connection ID field with an unpredictable
   value. This Destination Connection ID MUST be at least 8 bytes in
   length. Until a packet is received from the server, the client MUST
   use the same Destination Connection ID value on all packets in this
   connection.

   The Destination Connection ID field from the first Initial packet
   sent by a client is used to determine packet protection keys for
   Initial packets. These keys change after receiving a Retry packet;
   see Section 5.2 of [QUIC-TLS].

The client populates the Source Connection ID field with a value of its choosing and sets the Source Connection ID Length field to indicate the length.

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

Upon first receiving an Initial or Retry packet from the server, the client uses the Source Connection ID supplied by the server as the Destination Connection ID for subsequent packets, including any 0-RTT packets. This means that a client might have to change the connection ID it sets in the Destination Connection ID field twice during connection establishment: once in response to a Retry, and once in response to an Initial packet from the server. Once a client has received a valid Initial packet from the server, it MUST discard any subsequent packet it receives with a different Source Connection ID.

A client MUST change the Destination Connection ID it uses for sending packets in response to only the first received Initial or Retry packet. A server MUST set the Destination Connection ID it uses for sending packets based on the first received Initial packet. Any further changes to the Destination Connection ID are only permitted if the values are taken from NEW_CONNECTION_ID frames; if subsequent Initial packets include a different Source Connection ID, they MUST be discarded. This avoids unpredictable outcomes that might otherwise result from stateless processing of multiple Initial packets with different Source Connection IDs.

The Destination Connection ID that an endpoint sends can change over the lifetime of a connection, especially in response to connection migration (Section 9); see Section 5.1.1 for details.

7.3. Authenticating Connection IDs

The choice each endpoint makes about connection IDs during the handshake is authenticated by including all values in transport parameters; see Section 7.4. This ensures that all connection IDs used for the handshake are also authenticated by the cryptographic handshake.

Each endpoint includes the value of the Source Connection ID field from the first Initial packet it sent in the initial_source_connection_id transport parameter; see Section 18.2. A server includes the Destination Connection ID field from the first Initial packet it received from the client in the original_destination_connection_id transport parameter; if the server

sent a Retry packet, this refers to the first Initial packet received
before sending the Retry packet. If it sends a Retry packet, a
server also includes the Source Connection ID field from the Retry
packet in the retry_source_connection_id transport parameter.

The values provided by a peer for these transport parameters MUST
match the values that an endpoint used in the Destination and Source
Connection ID fields of Initial packets that it sent. Including
connection ID values in transport parameters and verifying them
ensures that that an attacker cannot influence the choice of
connection ID for a successful connection by injecting packets
carrying attacker-chosen connection IDs during the handshake.

An endpoint MUST treat absence of the initial_source_connection_id
transport parameter from either endpoint or absence of the
original_destination_connection_id transport parameter from the
server as a connection error of type TRANSPORT_PARAMETER_ERROR.

An endpoint MUST treat the following as a connection error of type
TRANSPORT_PARAMETER_ERROR or PROTOCOL_VIOLATION:

*   absence of the retry_source_connection_id transport parameter from
    the server after receiving a Retry packet,

*   presence of the retry_source_connection_id transport parameter
    when no Retry packet was received, or

*   a mismatch between values received from a peer in these transport
    parameters and the value sent in the corresponding Destination or
    Source Connection ID fields of Initial packets.

If a zero-length connection ID is selected, the corresponding
transport parameter is included with a zero-length value.

Figure 7 shows the connection IDs (with DCID=Destination Connection
ID, SCID=Source Connection ID) that are used in a complete handshake.
The exchange of Initial packets is shown, plus the later exchange of
1-RTT packets that includes the connection ID established during the
handshake.

```
Client                                                   Server

Initial: DCID=S1, SCID=C1 ->
                                    <- Initial: DCID=C1, SCID=S3
                          ...
1-RTT: DCID=S3 ->
                                           <- 1-RTT: DCID=C1
```

                   Figure 7: Use of Connection IDs in a Handshake

   Figure 8 shows a similar handshake that includes a Retry packet.

   Client   Server

   Initial: DCID=S1, SCID=C1 ->
                                         <- Retry: DCID=C1, SCID=S2
   Initial: DCID=S2, SCID=C1 ->
                                     <- Initial: DCID=C1, SCID=S3
                             ...
   1-RTT: DCID=S3 ->
                                             <- 1-RTT: DCID=C1

            Figure 8: Use of Connection IDs in a Handshake with Retry

   In both cases (Figure 7 and Figure 8), the client sets the value of
   the initial_source_connection_id transport parameter to "C1".

   When the handshake does not include a Retry (Figure 7), the server
   sets original_destination_connection_id to "S1" and
   initial_source_connection_id to "S3". In this case, the server does
   not include a retry_source_connection_id transport parameter.

   When the handshake includes a Retry (Figure 8), the server sets
   original_destination_connection_id to "S1",
   retry_source_connection_id to "S2", and initial_source_connection_id
   to "S3".

7.4. Transport Parameters

   During connection establishment, both endpoints make authenticated
   declarations of their transport parameters. Endpoints are required
   to comply with the restrictions that each parameter defines; the
   description of each parameter includes rules for its handling.

   Transport parameters are declarations that are made unilaterally by
   each endpoint. Each endpoint can choose values for transport
   parameters independent of the values chosen by its peer.

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

   QUIC includes the encoded transport parameters in the cryptographic
   handshake. Once the handshake completes, the transport parameters
   declared by the peer are available. Each endpoint validates the
   values provided by its peer.

Definitions for each of the defined transport parameters are included
in Section 18.2.

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

An endpoint MUST NOT send a parameter more than once in a given
transport parameters extension. An endpoint SHOULD treat receipt of
duplicate transport parameters as a connection error of type
TRANSPORT_PARAMETER_ERROR.

Endpoints use transport parameters to authenticate the negotiation of
connection IDs during the handshake; see Section 7.3.

Application Layer Protocol Negotiation (ALPN; see [ALPN]) allows
clients to offer multiple application protocols during connection
establishment. The transport parameters that a client includes
during the handshake apply to all application protocols that the
client offers. Application protocols can recommend values for
transport parameters, such as the initial flow control limits.
However, application protocols that set constraints on values for
transport parameters could make it impossible for a client to offer
multiple application protocols if these constraints conflict.

7.4.1. Values of Transport Parameters for 0-RTT

Using 0-RTT depends on both client and server using protocol
parameters that were negotiated from a previous connection. To
enable 0-RTT, endpoints store the value of the server transport
parameters from a connection and apply them to any 0-RTT packets that
are sent in subsequent connections to that peer. This information is
stored with any information required by the application protocol or
cryptographic handshake; see Section 4.6 of [QUIC-TLS].

Remembered transport parameters apply to the new connection until the
handshake completes and the client starts sending 1-RTT packets.
Once the handshake completes, the client uses the transport
parameters established in the handshake. Not all transport
parameters are remembered, as some do not apply to future connections
or they have no effect on use of 0-RTT.

The definition of a new transport parameter (Section 7.4.2) MUST
specify whether storing the transport parameter for 0-RTT is
mandatory, optional, or prohibited. A client need not store a
transport parameter it cannot process.

A client MUST NOT use remembered values for the following parameters:
ack_delay_exponent, max_ack_delay, initial_source_connection_id,
original_destination_connection_id, preferred_address,
retry_source_connection_id, and stateless_reset_token. The client
MUST use the server's new values in the handshake instead; if the
server does not provide new values, the default value is used.

A client that attempts to send 0-RTT data MUST remember all other
transport parameters used by the server that it is able to process.
The server can remember these transport parameters, or store an
integrity-protected copy of the values in the ticket and recover the
information when accepting 0-RTT data. A server uses the transport
parameters in determining whether to accept 0-RTT data.

If 0-RTT data is accepted by the server, the server MUST NOT reduce
any limits or alter any values that might be violated by the client
with its 0-RTT data. In particular, a server that accepts 0-RTT data
MUST NOT set values for the following parameters (Section 18.2) that
are smaller than the remembered value of the parameters.

*   active_connection_id_limit

*   initial_max_data

*   initial_max_stream_data_bidi_local

*   initial_max_stream_data_bidi_remote

*   initial_max_stream_data_uni

*   initial_max_streams_bidi

*   initial_max_streams_uni

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

A server MAY store and recover the previously sent values of the
max_idle_timeout, max_udp_payload_size, and disable_active_migration
parameters and reject 0-RTT if it selects smaller values. Lowering
the values of these parameters while also accepting 0-RTT data could
degrade the performance of the connection. Specifically, lowering
the max_udp_payload_size could result in dropped packets leading to
worse performance compared to rejecting 0-RTT data outright.

A server MUST reject 0-RTT data if the restored values for transport
parameters cannot be supported.

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

7.4.2. New Transport Parameters

New transport parameters can be used to negotiate new protocol
behavior. An endpoint MUST ignore transport parameters that it does
not support. Absence of a transport parameter therefore disables any
optional protocol feature that is negotiated using the parameter. As
described in Section 18.1, some identifiers are reserved in order to
exercise this requirement.

A client that does not understand a transport parameter can discard
it and attempt 0-RTT on subsequent connections. However, if the
client adds support for a discarded transport parameter, it risks
violating the constraints that the transport parameter establishes if
it attempts 0-RTT. New transport parameters can avoid this problem
by setting a default of the most conservative value.

New transport parameters can be registered according to the rules in
Section 22.2.

7.5. Cryptographic Message Buffering

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

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

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

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

8. Address Validation

Address validation ensures that an endpoint cannot be used for a
traffic amplification attack. In such an attack, a packet is sent to
a server with spoofed source address information that identifies a
victim. If a server generates more or larger packets in response to
that packet, the attacker can use the server to send more data toward
the victim than it would be able to send on its own.

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

8.1. Address Validation During Connection Establishment

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

Additionally, a server MAY consider the client address validated if
the client uses a connection ID chosen by the server and the
connection ID contains at least 64 bits of entropy.

Prior to validating the client address, servers MUST NOT send more
than three times as many bytes as the number of bytes they have
received. This limits the magnitude of any amplification attack that
can be mounted using spoofed source addresses. For the purposes of
avoiding amplification prior to address validation, servers MUST
count all of the payload bytes received in datagrams that are
uniquely attributed to a single connection. This includes datagrams
that contain packets that are successfully processed and datagrams
that contain packets that are all discarded.

Clients MUST ensure that UDP datagrams containing Initial packets
have UDP payloads of at least 1200 bytes, adding PADDING frames as
necessary. A client that sends padded datagrams allows the server to
send more data prior to completing address validation.

Loss of an Initial or Handshake packet from the server can cause a
deadlock if the client does not send additional Initial or Handshake
packets. A deadlock could occur when the server reaches its anti-
amplification limit and the client has received acknowledgements for
all the data it has sent. In this case, when the client has no
reason to send additional packets, the server will be unable to send
more data because it has not validated the client's address. To
prevent this deadlock, clients MUST send a packet on a probe timeout
(PTO, see Section 6.2 of [QUIC-RECOVERY]). Specifically, the client
MUST send an Initial packet in a UDP datagram that contains at least
1200 bytes if it does not have Handshake keys, and otherwise send a
Handshake packet.

A server might wish to validate the client address before starting
the cryptographic handshake. QUIC uses a token in the Initial packet
to provide address validation prior to completing the handshake.
This token is delivered to the client during connection establishment
with a Retry packet (see Section 8.1.2) or in a previous connection
using the NEW_TOKEN frame (see Section 8.1.3).

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

8.1.1. Token Construction

A token sent in a NEW_TOKEN frames or a Retry packet MUST be
constructed in a way that allows the server to identify how it was
provided to a client. These tokens are carried in the same field,
but require different handling from servers.

8.1.2. Address Validation using Retry Packets

   Upon receiving the client's Initial packet, the server can request
   address validation by sending a Retry packet (Section 17.2.5)
   containing a token. This token MUST be repeated by the client in all
   Initial packets it sends for that connection after it receives the
   Retry packet.

   In response to processing an Initial containing a token that was
   provided in a Retry packet, a server cannot send another Retry
   packet; it can only refuse the connection or permit it to proceed.

   As long as it is not possible for an attacker to generate a valid
   token for its own address (see Section 8.1.4) and the client is able
   to return that token, it proves to the server that it received the
   token.

   A server can also use a Retry packet to defer the state and
   processing costs of connection establishment. Requiring the server
   to provide a different connection ID, along with the
   original_destination_connection_id transport parameter defined in
   Section 18.2, forces the server to demonstrate that it, or an entity
   it cooperates with, received the original Initial packet from the
   client. Providing a different connection ID also grants a server
   some control over how subsequent packets are routed. This can be
   used to direct connections to a different server instance.

   If a server receives a client Initial that can be unprotected but
   contains an invalid Retry token, it knows the client will not accept
   another Retry token. The server can discard such a packet and allow
   the client to time out to detect handshake failure, but that could
   impose a significant latency penalty on the client. Instead, the
   server SHOULD immediately close (Section 10.2) the connection with an
   INVALID_TOKEN error. Note that a server has not established any
   state for the connection at this point and so does not enter the
   closing period.

   A flow showing the use of a Retry packet is shown in Figure 9.

```
Client                                                  Server

Initial[0]: CRYPTO[CH] ->

                                           <- Retry+Token

Initial+Token[1]: CRYPTO[CH] ->

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

                    Figure 9: Example Handshake with Retry

8.1.3. Address Validation for Future Connections

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

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

   Unlike the token that is created for a Retry packet, which is used
   immediately, the token sent in the NEW_TOKEN frame can be used after
   some period of time has passed. Thus, a token SHOULD have an
   expiration time, which could be either an explicit expiration time or
   an issued timestamp that can be used to dynamically calculate the
   expiration time. A server can store the expiration time or include
   it in an encrypted form in the token.

   A token issued with NEW_TOKEN MUST NOT include information that would
   allow values to be linked by an observer to the connection on which
   it was issued. For example, it cannot include the previous
   connection ID or addressing information, unless the values are
   encrypted. A server MUST ensure that every NEW_TOKEN frame it sends
   is unique across all clients, with the exception of those sent to
   repair losses of previously sent NEW_TOKEN frames. Information that
   allows the server to distinguish between tokens from Retry and
   NEW_TOKEN MAY be accessible to entities other than the server.

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

A token received in a NEW_TOKEN frame is applicable to any server
that the connection is considered authoritative for (e.g., server
names included in the certificate). When connecting to a server for
which the client retains an applicable and unused token, it SHOULD
include that token in the Token field of its Initial packet.
Including a token might allow the server to validate the client
address without an additional round trip. A client MUST NOT include
a token that is not applicable to the server that it is connecting
to, unless the client has the knowledge that the server that issued
the token and the server the client is connecting to are jointly
managing the tokens. A client MAY use a token from any previous
connection to that server.

A token allows a server to correlate activity between the connection
where the token was issued and any connection where it is used.
Clients that want to break continuity of identity with a server MAY
discard tokens provided using the NEW_TOKEN frame. In comparison, a
token obtained in a Retry packet MUST be used immediately during the
connection attempt and cannot be used in subsequent connection
attempts.

A client SHOULD NOT reuse a NEW_TOKEN token for different connection
attempts. Reusing a token allows connections to be linked by
entities on the network path; see Section 9.5.

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

When a server receives an Initial packet with an address validation
token, it MUST attempt to validate the token, unless it has already
completed address validation. If the token is invalid then the
server SHOULD proceed as if the client did not have a validated
address, including potentially sending a Retry. A server SHOULD
encode tokens provided with NEW_TOKEN frames and Retry packets
differently, and validate the latter more strictly. If the
validation succeeds, the server SHOULD then allow the handshake to
proceed.

   Note: The rationale for treating the client as unvalidated rather
      than discarding the packet is that the client might have received
      the token in a previous connection using the NEW_TOKEN frame, and
      if the server has lost state, it might be unable to validate the
      token at all, leading to connection failure if the packet is
      discarded.

   In a stateless design, a server can use encrypted and authenticated
   tokens to pass information to clients that the server can later
   recover and use to validate a client address. Tokens are not
   integrated into the cryptographic handshake and so they are not
   authenticated. For instance, a client might be able to reuse a
   token. To avoid attacks that exploit this property, a server can
   limit its use of tokens to only the information needed to validate
   client addresses.

   Clients MAY use tokens obtained on one connection for any connection
   attempt using the same version. When selecting a token to use,
   clients do not need to consider other properties of the connection
   that is being attempted, including the choice of possible application
   protocols, session tickets, or other connection properties.

8.1.4. Address Validation Token Integrity

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

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

   There is no need for a single well-defined format for the token
   because the server that generates the token also consumes it. Tokens
   sent in Retry packets SHOULD include information that allows the
   server to verify that the source IP address and port in client
   packets remain constant.

Tokens sent in NEW_TOKEN frames MUST include information that allows
the server to verify that the client IP address has not changed from
when the token was issued. Servers can use tokens from NEW_TOKEN in
deciding not to send a Retry packet, even if the client address has
changed. If the client IP address has changed, the server MUST
adhere to the anti-amplification limits found in Section 8.1. Note
that in the presence of NAT, this requirement might be insufficient
to protect other hosts that share the NAT from amplification attack.

Attackers could replay tokens to use servers as amplifiers in DDoS
attacks. To protect against such attacks, servers MUST ensure that
replay of tokens is prevented or limited. Servers SHOULD ensure that
tokens sent in Retry packets are only accepted for a short time.
Tokens that are provided in NEW_TOKEN frames (Section 19.7) need to
be valid for longer, but SHOULD NOT be accepted multiple times in a
short period. Servers are encouraged to allow tokens to be used only
once, if possible; tokens MAY include additional information about
clients to further narrow applicability or reuse.

8.2. Path Validation

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

Path validation tests that packets sent on a path to a peer are
received by that peer. Path validation is used to ensure that
packets received from a migrating peer do not carry a spoofed source
address.

Path validation does not validate that a peer can send in the return
direction. Acknowledgments cannot be used for return path validation
because they contain insufficient entropy and might be spoofed.
Endpoints independently determine reachability on each direction of a
path, and therefore return reachability can only be established by
the peer.

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 include other frames with the PATH_CHALLENGE and
PATH_RESPONSE frames used for path validation. In particular, an
endpoint can include PADDING frames with a PATH_CHALLENGE frame for
Path Maximum Transmission Unit Discovery (PMTUD; see Section 14.2.1);
it can also include its own PATH_CHALLENGE frame with a PATH_RESPONSE
frame.

An endpoint uses a new connection ID for probes sent from a new local
address; see Section 9.5. When probing a new path, an endpoint can
ensure that its peer has an unused connection ID available for
responses. Sending NEW_CONNECTION_ID and PATH_CHALLENGE frames in
the same packet, if the peer's active_connection_id_limit permits,
ensures that an unused connection ID will be available to the peer
when sending a response.

An endpoint can choose to simultaneously probe multiple paths. The
number of simultaneous paths used for probes is limited by the number
of extra connection IDs its peer has previously supplied, since each
new local address used for a probe requires a previously unused
connection ID.

8.2.1. Initiating Path Validation

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

An endpoint MAY send multiple PATH_CHALLENGE frames to guard against
packet loss. However, an endpoint SHOULD NOT send multiple
PATH_CHALLENGE frames in a single packet.

An endpoint SHOULD NOT probe a new path with packets containing a
PATH_CHALLENGE frame more frequently than it would send an Initial
packet. This ensures that connection migration is no more load on a
new path than establishing a new connection.

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

An endpoint MUST expand datagrams that contain a PATH_CHALLENGE frame
to at least the smallest allowed maximum datagram size of 1200 bytes.
Sending UDP datagrams of this size ensures that the network path from
the endpoint to the peer can be used for QUIC; see Section 14.

8.2.2. Path Validation Responses

On receiving a PATH_CHALLENGE frame, an endpoint MUST respond by
echoing the data contained in the PATH_CHALLENGE frame in a
PATH_RESPONSE frame. An endpoint MUST NOT delay transmission of a
packet containing a PATH_RESPONSE frame unless constrained by
congestion control.

A PATH_RESPONSE frame MUST be sent on the network path where the
PATH_CHALLENGE was received. This ensures that path validation by a
peer only succeeds if the path is functional in both directions.
This requirement MUST NOT be enforced by the endpoint that initiates
path validation as that would enable an attack on migration; see
Section 9.3.3.

An endpoint MUST expand datagrams that contain a PATH_RESPONSE frame
to at least the smallest allowed maximum datagram size of 1200 bytes.
This verifies that the path is able to carry datagrams of this size
in both directions.

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

8.2.3. Successful Path Validation

Path validation succeeds when a PATH_RESPONSE frame is received that
contains the data that was sent in a previous PATH_CHALLENGE frame.
A PATH_RESPONSE frame received on any network path validates the path
on which the PATH_CHALLENGE was sent.

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.

8.2.4. Failed Path Validation

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

Endpoints SHOULD abandon path validation based on a timer. When
setting this timer, implementations are cautioned that the new path
could have a longer round-trip time than the original. A value of
three times the larger of the current Probe Timeout (PTO) or the
initial timeout (that is, 2*kInitialRtt) as defined in
[QUIC-RECOVERY] is RECOMMENDED. That is:

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

This timeout allows for multiple PTOs to expire prior to failing path
validation, so that loss of a single PATH_CHALLENGE or PATH_RESPONSE
frame does not cause path validation failure.

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

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

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

9. Connection Migration

The use of a connection ID allows connections to survive changes to
endpoint addresses (IP address and port), such as those caused by an
endpoint migrating to a new network. This section describes the
process by which an endpoint migrates to a new address.

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

If the peer sent the disable_active_migration transport parameter, an
endpoint also MUST NOT send packets (including probing packets; see
Section 9.1) from a different local address to the address the peer
used during the handshake, unless the endpoint has acted on a
preferred_address transport parameter from the peer. If the peer
violates this requirement, the endpoint MUST either drop the incoming
packets on that path without generating a stateless reset or proceed
with path validation and allow the peer to migrate. Generating a

stateless reset or closing the connection would allow third parties
in the network to cause connections to close by spoofing or otherwise
manipulating observed traffic.

Not all changes of peer address are intentional, or active,
migrations. The peer could experience NAT rebinding: a change of
address due to a middlebox, usually a NAT, allocating a new outgoing
port or even a new outgoing IP address for a flow. An endpoint MUST
perform path validation (Section 8.2) if it detects any change to a
peer's address, unless it has previously validated that address.

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

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

9.1. Probing a New Path

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

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

9.2. Initiating Connection Migration

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

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

   To establish reachability on the new path, an endpoint initiates path
   validation (Section 8.2) on the new path. An endpoint MAY defer path
   validation until after a peer sends the next non-probing frame to its
   new address.

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

   The new path might not have the same ECN capability. Therefore, the
   endpoint validates ECN capability as described in Section 13.4.

9.3. Responding to Connection Migration

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

   If the recipient permits the migration, it MUST send subsequent
   packets to the new peer address and MUST initiate path validation
   (Section 8.2) to verify the peer's ownership of the address if
   validation is not already underway.

   An endpoint only changes the address to which it sends packets 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.

   An endpoint MAY send data to an unvalidated peer address, but it MUST
   protect against potential attacks as described in Section 9.3.1 and
   Section 9.3.2. An endpoint MAY skip validation of a peer address if
   that address has been seen recently. In particular, if an endpoint
   returns to a previously-validated path after detecting some form of
   spurious migration, skipping address validation and restoring loss
   detection and congestion state can reduce the performance impact of
   the attack.

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

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

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

9.3.1. Peer Address Spoofing

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

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

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

9.3.2. On-Path Address Spoofing

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

To protect the connection from failing due to such a spurious
migration, an endpoint MUST revert to using the last validated peer
address when validation of a new peer address fails. Additionally,
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, thus containing migrations initiated by the attacker
injecting a single packet.

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.

9.3.3. Off-Path Packet Forwarding

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

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

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

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

An endpoint that receives a PATH_CHALLENGE on an active path SHOULD send a non-probing packet in response. If the non-probing packet arrives before any copy made by an attacker, this results in the connection being migrated back to the original path. Any subsequent migration to another path restarts this entire process.

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

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

9.4. Loss Detection and Congestion Control

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

   On confirming a peer's ownership of its new address, an endpoint MUST
   immediately reset the congestion controller and round-trip time
   estimator for the new path to initial values (see Appendices A.3 and
   B.3 in [QUIC-RECOVERY]) unless the only change in the peer's address
   is its port number. Because port-only changes are commonly the
   result of NAT rebinding or other middlebox activity, the endpoint MAY
   instead retain its congestion control state and round-trip estimate
   in those cases instead of reverting to initial values. In cases
   where congestion control state retained from an old path is used on a
   new path with substantially different characteristics, a sender may
   transmit too aggressively until the congestion controller and the RTT
   estimator have adapted. Generally, implementations are advised to be
   cautious when using previous values on a new path.

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

   While multiple paths might be used during connection migration, a
   single congestion control context and a single loss recovery context
   (as described in [QUIC-RECOVERY]) may be adequate. For instance, an
   endpoint might delay switching to a new congestion control context
   until it is confirmed that an old path is no longer needed (such as
   the case in Section 9.3.3).

   A sender can make exceptions for probe packets so that their loss
   detection is independent and does not unduly cause the congestion
   controller to reduce its sending rate. An endpoint might set a
   separate timer when a PATH_CHALLENGE is sent, which is cancelled if
   the corresponding PATH_RESPONSE is received. If the timer fires
   before the PATH_RESPONSE is received, the endpoint might send a new
   PATH_CHALLENGE, and restart the timer for a longer period of time.
   This timer SHOULD be set as described in Section 6.2.1 of
   [QUIC-RECOVERY] and MUST NOT be more aggressive.

9.5. Privacy Implications of Connection Migration

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

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

   An endpoint MUST NOT reuse a connection ID when sending from more
   than one local address, for example when initiating connection
   migration as described in Section 9.2 or when probing a new network
   path as described in Section 9.1.

   Similarly, an endpoint MUST NOT reuse a connection ID when sending to
   more than one destination address. Due to network changes outside
   the control of its peer, an endpoint might receive packets from a new
   source address with the same destination connection ID, in which case
   it MAY continue to use the current connection ID with the new remote
   address while still sending from the same local address.

   These requirements regarding connection ID reuse apply only to the
   sending of packets, as unintentional changes in path without a change
   in connection ID are possible. For example, after a period of
   network inactivity, NAT rebinding might cause packets to be sent on a
   new path when the client resumes sending. An endpoint responds to
   such an event as described in Section 9.3.

Using different connection IDs for packets sent in both directions on
each new network path eliminates the use of the connection ID for
linking packets from the same connection across different network
paths. Header protection ensures that packet numbers cannot be used
to correlate activity. This does not prevent other properties of
packets, such as timing and size, from being used to correlate
activity.

An endpoint SHOULD NOT initiate migration with a peer that has
requested a zero-length connection ID, because traffic over the new
path might be trivially linkable to traffic over the old one. If the
server is able to associate packets with a zero-length connection ID
to the right connection, it means that the server is using other
information to demultiplex packets. For example, a server might
provide a unique address to every client, for instance using HTTP
alternative services [ALTSVC]. Information that might allow correct
routing of packets across multiple network paths will also allow
activity on those paths to be linked by entities other than the peer.

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 do not experience NAT rebindings
or genuine migrations. Changing port number can cause a peer to
reset its congestion state (see Section 9.4), so the port SHOULD only
be changed infrequently.

An endpoint that exhausts available connection IDs cannot probe new
paths or initiate migration, nor can it respond to probes or attempts
by its peer to migrate. To ensure that migration is possible and
packets sent on different paths cannot be correlated, endpoints
SHOULD provide new connection IDs before peers migrate; see
Section 5.1.1. If a peer might have exhausted available connection
IDs, a migrating endpoint could include a NEW_CONNECTION_ID frame in
all packets sent on a new network path.

9.6. Server's Preferred Address

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

Migrating a connection to a new server address mid-connection is not
supported by the version of QUIC specified in this document. If a
client receives packets from a new server address when the client has
not initiated a migration to that address, the client SHOULD discard
these packets.

9.6.1. Communicating a Preferred Address

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

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

Once the handshake is confirmed, the client SHOULD select one of the
two addresses provided by the server and initiate path validation
(see Section 8.2). A client constructs packets using any previously
unused active connection ID, taken from either the preferred_address
transport parameter or a NEW_CONNECTION_ID frame.

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

9.6.2. Migration to a Preferred Address

A client that migrates to a preferred address MUST validate the
address it chooses before migrating; see Section 21.4.3.

A server might receive a packet addressed to its preferred IP address
at any time after it accepts a connection. If this packet contains a
PATH_CHALLENGE frame, the server sends a packet containing a
PATH_RESPONSE frame as per Section 8.2. The server MUST send non-
probing packets from its original address until it receives a non-
probing packet from the client at its preferred address and until the
server has validated the new path.

The server MUST probe on the path toward the client from its
preferred address. This helps to guard against spurious migration
initiated by an attacker.

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

The addresses that a server provides in the preferred_address
transport parameter are only valid for the connection in which they
are provided. A client MUST NOT use these for other connections,
including connections that are resumed from the current connection.

9.6.3. Interaction of Client Migration and Preferred Address

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

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

If packets received at the server's preferred address have a
different source address than observed from the client during the
handshake, the server MUST protect against potential attacks as
described in Section 9.3.1 and Section 9.3.2. In addition to
intentional simultaneous migration, this might also occur because the
client's access network used a different NAT binding for the server's
preferred address.

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

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

The connection ID provided in the preferred_address transport
parameter is not specific to the addresses that are provided. This
connection ID is provided to ensure that the client has a connection
ID available for migration, but the client MAY use this connection ID
on any path.

9.7. Use of IPv6 Flow-Label and Migration

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

   The IPv6 flow label SHOULD be a pseudo-random function of the source
   and destination addresses, source and destination UDP ports, and the
   Destination Connection ID field. The flow label generation MUST be
   designed to minimize the chances of linkability with a previously
   used flow label, as this would enable correlating activity on
   multiple paths; see Section 9.5.

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

10. Connection Termination

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

   * idle timeout (Section 10.1)

   * immediate close (Section 10.2)

   * stateless reset (Section 10.3)

   An endpoint MAY discard connection state if it does not have a
   validated path on which it can send packets; see Section 8.2.

10.1. Idle Timeout

   If a max_idle_timeout is specified by either peer in its transport
   parameters (Section 18.2), the connection is silently closed and its
   state is discarded when it remains idle for longer than the minimum
   of both peers max_idle_timeout values.

   Each endpoint advertises a max_idle_timeout, but the effective value
   at an endpoint is computed as the minimum of the two advertised
   values. By announcing a max_idle_timeout, an endpoint commits to
   initiating an immediate close (Section 10.2) if it abandons the
   connection prior to the effective value.

   An endpoint restarts its idle timer when a packet from its peer is
   received and processed successfully. An endpoint also restarts its
   idle timer when sending an ack-eliciting packet if no other ack-

eliciting packets have been sent since last receiving and processing
a packet. Restarting this timer when sending a packet ensures that
connections are not closed after new activity is initiated.

To avoid excessively small idle timeout periods, endpoints MUST
increase the idle timeout period to be at least three times the
current Probe Timeout (PTO). This allows for multiple PTOs to
expire, and therefore multiple probes to be sent and lost, prior to
idle timeout.

10.1.1. Liveness Testing

An endpoint that sends packets close to the effective timeout risks
having them be discarded at the peer, since the idle timeout period
might have expired at the peer before these packets arrive.

An endpoint can send a PING or another ack-eliciting frame to test
the connection for liveness if the peer could time out soon, such as
within a PTO; see Section 6.2 of [QUIC-RECOVERY]. This is especially
useful if any available application data cannot be safely retried.
Note that the application determines what data is safe to retry.

10.1.2. Deferring Idle Timeout

An endpoint might need to send ack-eliciting packets to avoid an idle
timeout if it is expecting response data, but does not have or is
unable to send application data.

An implementation of QUIC might provide applications with an option
to defer an idle timeout. This facility could be used when the
application wishes to avoid losing state that has been associated
with an open connection, but does not expect to exchange application
data for some time. With this option, an endpoint could send a PING
frame (Section 19.2) periodically, which will cause the peer to
restart its idle timeout period. Sending a packet containing a PING
frame restarts the idle timeout for this endpoint also if this is the
first ack-eliciting packet sent since receiving a packet. Sending a
PING frame causes the peer to respond with an acknowledgment, which
also restarts the idle timeout for the endpoint.

Application protocols that use QUIC SHOULD provide guidance on when
deferring an idle timeout is appropriate. Unnecessary sending of
PING frames could have a detrimental effect on performance.

A connection will time out if no packets are sent or received for a
period longer than the time negotiated using the max_idle_timeout
transport parameter; see Section 10. However, state in middleboxes
might time out earlier than that. Though REQ-5 in [RFC4787]

recommends a 2 minute timeout interval, experience shows that sending
packets every 30 seconds is necessary to prevent the majority of
middleboxes from losing state for UDP flows [GATEWAY].

10.2. Immediate Close

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

After sending a CONNECTION_CLOSE frame, an endpoint immediately
enters the closing state; see Section 10.2.1. After receiving a
CONNECTION_CLOSE frame, endpoints enter the draining state; see
Section 10.2.2.

Violations of the protocol lead to an immediate close.

An immediate close can be used after an application protocol has
arranged to close a connection. This might be after the application
protocol negotiates a graceful shutdown. The application protocol
can exchange messages that are needed for both application endpoints
to agree that the connection can be closed, after which the
application requests that QUIC close the connection. When QUIC
consequently closes the connection, a CONNECTION_CLOSE frame with an
application-supplied error code will be used to signal closure to the
peer.

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

Disposing of connection state prior to exiting the closing or
draining state could result in an endpoint generating a stateless
reset unnecessarily when it receives a late-arriving packet.
Endpoints that have some alternative means to ensure that late-
arriving packets do not induce a response, such as those that are
able to close the UDP socket, MAY end these states earlier to allow
for faster resource recovery. Servers that retain an open socket for
accepting new connections SHOULD NOT end the closing or draining
states early.

Once its closing or draining state ends, an endpoint SHOULD discard
all connection state. The endpoint MAY send a stateless reset in
response to any further incoming packets belonging to this
connection.

10.2.1. Closing Connection State

   An endpoint enters the closing state after initiating an immediate
   close.

   In the closing state, an endpoint retains only enough information to
   generate a packet containing a CONNECTION_CLOSE frame and to identify
   packets as belonging to the connection. An endpoint in the closing
   state sends a packet containing a CONNECTION_CLOSE frame in response
   to any incoming packet that it attributes to the connection.

   An endpoint SHOULD limit the rate at which it generates packets in
   the closing state. For instance, an endpoint could wait for a
   progressively increasing number of received packets or amount of time
   before responding to received packets.

   An endpoint's selected connection ID and the QUIC version are
   sufficient information to identify packets for a closing connection;
   the endpoint MAY discard all other connection state. An endpoint
   that is closing is not required to process any received frame. An
   endpoint MAY retain packet protection keys for incoming packets to
   allow it to read and process a CONNECTION_CLOSE frame.

   An endpoint MAY drop packet protection keys when entering the closing
   state and send a packet containing a CONNECTION_CLOSE frame in
   response to any UDP datagram that is received. However, an endpoint
   that discards packet protection keys cannot identify and discard
   invalid packets. To avoid being used for an amplication attack, such
   endpoints MUST limit the cumulative size of packets it sends to three
   times the cumulative size of the packets that are received and
   attributed to the connection. To minimize the state that an endpoint
   maintains for a closing connection, endpoints MAY send the exact same
   packet in response to any received packet.

   Note: Allowing retransmission of a closing packet is an exception to
      the requirement that a new packet number be used for each packet
      in Section 12.3. 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.

   While in the closing state, an endpoint could receive packets from a
   new source address, possibly indicating a connection migration; see
   Section 9. An endpoint in the closing state MUST either discard
   packets received from an unvalidated address or limit the cumulative
   size of packets it sends to an unvalidated address to three times the
   size of packets it receives from that address.

An endpoint is not expected to handle key updates when it is closing
(Section 6 of [QUIC-TLS]). A key update might prevent the endpoint
from moving from the closing state to the draining state, as the
endpoint will not be able to process subsequently received packets,
but it otherwise has no impact.

10.2.2. Draining Connection State

The draining state is entered once an endpoint receives a
CONNECTION_CLOSE frame, which indicates 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 that receives a CONNECTION_CLOSE frame MAY send a single
packet containing a CONNECTION_CLOSE frame before entering the
draining state, using a NO_ERROR code if appropriate. An endpoint
MUST NOT send further packets. Doing so could result in a constant
exchange of CONNECTION_CLOSE frames until one of the endpoints exits
the closing state.

An endpoint MAY enter the draining state from the closing state if it
receives a CONNECTION_CLOSE frame, which indicates that the peer is
also closing or draining. In this case, the draining state SHOULD
end when the closing state would have ended. In other words, the
endpoint uses the same end time, but ceases transmission of any
packets on this connection.

10.2.3. Immediate Close During the Handshake

When sending CONNECTION_CLOSE, the goal is to ensure that the peer
will process the frame. Generally, this means sending the frame in a
packet with the highest level of packet protection to avoid the
packet being discarded. After the handshake is confirmed (see
Section 4.1.2 of [QUIC-TLS]), an endpoint MUST send any
CONNECTION_CLOSE frames in a 1-RTT packet. However, prior to
confirming the handshake, it is possible that more advanced packet
protection keys are not available to the peer, so another
CONNECTION_CLOSE frame MAY be sent in a packet that uses a lower
packet protection level. More specifically:

* A client will always know whether the server has Handshake keys
   (see Section 17.2.2.1), but it is possible that a server does not
   know whether the client has Handshake keys. Under these
   circumstances, a server SHOULD send a CONNECTION_CLOSE frame in
   both Handshake and Initial packets to ensure that at least one of
   them is processable by the client.

   * A client that sends CONNECTION_CLOSE in a 0-RTT packet cannot be
      assured that the server has accepted 0-RTT. Sending a
      CONNECTION_CLOSE frame in an Initial packet makes it more likely
      that the server can receive the close signal, even if the
      application error code might not be received.

   * Prior to confirming the handshake, a peer might be unable to
      process 1-RTT packets, so an endpoint SHOULD send CONNECTION_CLOSE
      in both Handshake and 1-RTT packets. A server SHOULD also send
      CONNECTION_CLOSE in an Initial packet.

   Sending a CONNECTION_CLOSE of type 0x1d in an Initial or Handshake
   packet could expose application state or be used to alter application
   state. A CONNECTION_CLOSE of type 0x1d MUST be replaced by a
   CONNECTION_CLOSE of type 0x1c when sending the frame in Initial or
   Handshake packets. Otherwise, information about the application
   state might be revealed. Endpoints MUST clear the value of the
   Reason Phrase field and SHOULD use the APPLICATION_ERROR code when
   converting to a CONNECTION_CLOSE of type 0x1c.

   CONNECTION_CLOSE frames sent in multiple packet types can be
   coalesced into a single UDP datagram; see Section 12.2.

   An endpoint can send a CONNECTION_CLOSE frame in an Initial packet.
   This might be in response to unauthenticated information received in
   Initial or Handshake packets. Such an immediate close might expose
   legitimate connections to a denial of service. QUIC does not include
   defensive measures for on-path attacks during the handshake; see
   Section 21.1. However, at the cost of reducing feedback about errors
   for legitimate peers, some forms of denial of service can be made
   more difficult for an attacker if endpoints discard illegal packets
   rather than terminating a connection with CONNECTION_CLOSE. For this
   reason, endpoints MAY discard packets rather than immediately close
   if errors are detected in packets that lack authentication.

   An endpoint that has not established state, such as a server that
   detects an error in an Initial packet, does not enter the closing
   state. An endpoint that has no state for the connection does not
   enter a closing or draining period on sending a CONNECTION_CLOSE
   frame.

10.3. Stateless Reset

   A stateless reset is provided as an option of last resort for an
   endpoint that does not have access to the state of a connection. A
   crash or outage might result in peers continuing to send data to an
   endpoint that is unable to properly continue the connection. An
   endpoint MAY send a stateless reset in response to receiving a packet
   that it cannot associate with an active connection.

   A stateless reset is not appropriate for indicating errors in active
   connections. An endpoint that wishes to communicate a fatal
   connection error MUST use a CONNECTION_CLOSE frame if it is able.

   To support this process, a token is sent by endpoints. The token is
   carried in the Stateless Reset Token field of a NEW_CONNECTION_ID
   frame. Servers can also specify a stateless_reset_token transport
   parameter during the handshake that applies to the connection ID that
   it selected during the handshake; clients cannot use this transport
   parameter because their transport parameters do not have
   confidentiality protection. These tokens are protected by
   encryption, so only client and server know their value. Tokens are
   invalidated when their associated connection ID is retired via a
   RETIRE_CONNECTION_ID frame (Section 19.16).

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

   Stateless Reset {
     Fixed Bits (2) = 1,
     Unpredictable Bits (38..),
     Stateless Reset Token (128),
   }

                      Figure 10: Stateless Reset Packet

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

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

To entities other than its intended recipient, a stateless reset will appear to be a packet with a short header. For the stateless reset to appear as a valid QUIC packet, the Unpredictable Bits field needs to include at least 38 bits of data (or 5 bytes, less the two fixed bits).

The resulting minimum size of 21 bytes does not guarantee that a stateless reset is difficult to distinguish from other packets if the recipient requires the use of a connection ID. To achieve that end, the endpoint SHOULD ensure that all packets it sends are at least 22 bytes longer than the minimum connection ID length that it requests the peer to include in its packets, adding PADDING frames as necessary. This ensures that any stateless reset sent by the peer is indistinguishable from a valid packet sent to the endpoint. An endpoint that sends a stateless reset in response to a packet that is 43 bytes or shorter SHOULD send a stateless reset that is one byte shorter than the packet it responds to.

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

An endpoint MUST NOT send a stateless reset that is three times or more larger than the packet it receives to avoid being used for amplification. Section 10.3.3 describes additional limits on stateless reset size.

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

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

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

An endpoint cannot determine the Source Connection ID from a packet
with a short header, therefore it cannot set the Destination
Connection ID in the stateless reset packet. The Destination
Connection ID will therefore differ from the value used in previous
packets. A random Destination Connection ID makes the connection ID
appear to be the result of moving to a new connection ID that was
provided using a NEW_CONNECTION_ID frame (Section 19.15).

Using a randomized connection ID results in two problems:

* The packet might not reach the peer. If the Destination
   Connection ID is critical for routing toward the peer, then this
   packet could be incorrectly routed. This might also trigger
   another Stateless Reset in response; see Section 10.3.3. A
   Stateless Reset that is not correctly routed is an ineffective
   error detection and recovery mechanism. In this case, endpoints
   will need to rely on other methods - such as timers - to detect
   that the connection has failed.

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

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

10.3.1. Detecting a Stateless Reset

An endpoint detects a potential stateless reset using the trailing 16
bytes of the UDP datagram. An endpoint remembers all Stateless Reset
Tokens associated with the connection IDs and remote addresses for
datagrams it has recently sent. This includes Stateless Reset Tokens
from NEW_CONNECTION_ID frames and the server's transport parameters
but excludes Stateless Reset Tokens associated with connection IDs
that are either unused or retired. The endpoint identifies a
received datagram as a stateless reset by comparing the last 16 bytes
of the datagram with all Stateless Reset Tokens associated with the
remote address on which the datagram was received.

This comparison can be performed for every inbound datagram.
Endpoints MAY skip this check if any packet from a datagram is
successfully processed. However, the comparison MUST be performed
when the first packet in an incoming datagram either cannot be
associated with a connection, or cannot be decrypted.

An endpoint MUST NOT check for any Stateless Reset Tokens associated
with connection IDs it has not used or for connection IDs that have
been retired.

When comparing a datagram to Stateless Reset Token values, endpoints
MUST perform the comparison without leaking information about the
value of the token. For example, performing this comparison in
constant time protects the value of individual Stateless Reset Tokens
from information leakage through timing side channels. Another
approach would be to store and compare the transformed values of
Stateless Reset Tokens instead of the raw token values, where the
transformation is defined as a cryptographically-secure pseudo-random
function using a secret key (e.g., block cipher, HMAC [RFC2104]). An
endpoint is not expected to protect information about whether a
packet was successfully decrypted, or the number of valid Stateless
Reset Tokens.

If the last 16 bytes of the datagram are identical in value to a
Stateless Reset Token, the endpoint MUST enter the draining period
and not send any further packets on this connection.

10.3.2. Calculating a Stateless Reset Token

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

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

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

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

Revealing the Stateless Reset Token allows any entity to terminate
the connection, so a value can only be used once. This method for
choosing the Stateless Reset Token means that the combination of
connection ID and static key MUST NOT be used for another connection.
A denial of service attack is possible if the same connection ID is
used by instances that share a static key, or if an attacker can
cause a packet to be routed to an instance that has no state but the
same static key; see Section 21.10. A connection ID from a
connection that is reset by revealing the Stateless Reset Token MUST
NOT be reused for new connections at nodes that share a static key.

The same Stateless Reset Token MUST NOT be used for multiple
connection IDs. Endpoints are not required to compare new values
against all previous values, but a duplicate value MAY be treated as
a connection error of type PROTOCOL_VIOLATION.

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

10.3.3. Looping

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

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

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

Reducing the size of a Stateless Reset below 41 bytes means that the
packet could reveal to an observer that it is a Stateless Reset,
depending upon the length of the peer's connection IDs. Conversely,
refusing to send a Stateless Reset in response to a small packet
might result in Stateless Reset not being useful in detecting cases
of broken connections where only very small packets are sent; such
failures might only be detected by other means, such as timers.

11. Error Handling

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

The most appropriate error code (Section 20) SHOULD be included in
the frame that signals the error. Where this specification
identifies error conditions, it also identifies the error code that
is used; though these are worded as requirements, different
implementation strategies might lead to different errors being
reported. In particular, an endpoint MAY use any applicable error
code when it detects an error condition; a generic error code (such
as PROTOCOL_VIOLATION or INTERNAL_ERROR) can always be used in place
of specific error codes.

A stateless reset (Section 10.3) is not suitable for any error that
can be signaled with a CONNECTION_CLOSE or RESET_STREAM frame. A
stateless reset MUST NOT be used by an endpoint that has the state
necessary to send a frame on the connection.

11.1. Connection Errors

Errors that result in the connection being unusable, such as an
obvious violation of protocol semantics or corruption of state that
affects an entire connection, MUST be signaled using a
CONNECTION_CLOSE frame (Section 19.19).

Application-specific protocol errors are signaled using the
CONNECTION_CLOSE frame with a frame type of 0x1d. Errors that are
specific to the transport, including all those described in this
document, are carried in the CONNECTION_CLOSE frame with a frame type
of 0x1c.

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

An endpoint that chooses not to retransmit packets containing a
CONNECTION_CLOSE frame risks a peer missing the first such packet.
The only mechanism available to an endpoint that continues to receive
data for a terminated connection is to use the stateless reset
process (Section 10.3).

11.2. Stream Errors

If an application-level error affects a single stream, but otherwise
leaves the connection in a recoverable state, the endpoint can send a
RESET_STREAM frame (Section 19.4) with an appropriate error code to
terminate just the affected stream.

Resetting a stream without the involvement of the application
protocol could cause the application protocol to enter an
unrecoverable state. RESET_STREAM MUST only be instigated by the
application protocol that uses QUIC.

The semantics of the application error code carried in RESET_STREAM
are defined by the application protocol. Only the application
protocol is able to cause a stream to be terminated. A local
instance of the application protocol uses a direct API call and a
remote instance uses the STOP_SENDING frame, which triggers an
automatic RESET_STREAM.

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

12. Packets and Frames

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

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

Packets with the short header are designed for minimal overhead and
are used after a connection is established and 1-RTT keys are
available; see Section 17.3.

12.1. Protected Packets

QUIC packets have different levels of cryptographic protection based
on the type of packet. Details of packet protection are found in
[QUIC-TLS]; this section includes an overview of the protections that
are provided.

Version Negotiation packets have no cryptographic protection; see
[QUIC-INVARIANTS].

Retry packets use an authenticated encryption with associated data
function (AEAD; [AEAD]) to protect against accidental modification.

Initial packets use an AEAD, the keys for which are derived using a
value that is visible on the wire. Initial packets therefore do not
have effective confidentiality protection. Initial protection exists
to ensure that the sender of the packet is on the network path. Any
entity that receives an Initial packet from a client can recover the
keys that will allow them to both read the contents of the packet and
generate Initial packets that will be successfully authenticated at
either endpoint.

All other packets are protected with keys derived from the
cryptographic handshake. The cryptographic handshake ensures that
only the communicating endpoints receive the corresponding keys for
Handshake, 0-RTT, and 1-RTT packets. Packets protected with 0-RTT
and 1-RTT keys have strong confidentiality and integrity protection.

The Packet Number field that appears in some packet types has
alternative confidentiality protection that is applied as part of
header protection; see Section 5.4 of [QUIC-TLS] for details. The
underlying packet number increases with each packet sent in a given
packet number space; see Section 12.3 for details.

12.2. Coalescing Packets

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

   Using the Length field, a sender can coalesce multiple QUIC packets
   into one UDP datagram. This can reduce the number of UDP datagrams
   needed to complete the cryptographic handshake and start sending
   data. This can also be used to construct PMTU probes; see
   Section 14.4.1. Receivers MUST be able to process coalesced packets.

   Coalescing packets in order of increasing encryption levels (Initial,
   0-RTT, Handshake, 1-RTT; see Section 4.1.4 of [QUIC-TLS]) makes it
   more likely the receiver will be able to process all the packets in a
   single pass. A packet with a short header does not include a length,
   so it can only be the last packet included in a UDP datagram. An
   endpoint SHOULD include multiple frames in a single packet if they
   are to be sent at the same encryption level, instead of coalescing
   multiple packets at the same encryption level.

   Receivers MAY route based on the information in the first packet
   contained in a UDP datagram. Senders MUST NOT coalesce QUIC packets
   with different connection IDs into a single UDP datagram. Receivers
   SHOULD ignore any subsequent packets with a different Destination
   Connection ID than the first packet in the datagram.

   Every QUIC packet that is coalesced into a single UDP datagram is
   separate and complete. The receiver of coalesced QUIC packets MUST
   individually process each QUIC packet and separately acknowledge
   them, as if they were received as the payload of different UDP
   datagrams. For example, if decryption fails (because the keys are
   not available or any other reason), the receiver MAY either discard
   or buffer the packet for later processing and MUST attempt to process
   the remaining packets.

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

12.3. Packet Numbers

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

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

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

   Packet numbers are divided into 3 spaces in QUIC:

   * Initial space: All Initial packets (Section 17.2.2) are in this
      space.

   * Handshake space: All Handshake packets (Section 17.2.4) are in
      this space.

   * Application data space: All 0-RTT (Section 17.2.3) and 1-RTT
      (Section 17.3) encrypted packets are in this space.

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

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

   This enforces cryptographic separation between the data sent in the
   different packet number spaces. Packet numbers in each space start
   at packet number 0. Subsequent packets sent in the same packet
   number space MUST increase the packet number by at least one.

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

A QUIC endpoint MUST NOT reuse a packet number within the same packet
number space in one connection. If the packet number for sending
reaches 2^62 - 1, the sender MUST close the connection without
sending a CONNECTION_CLOSE frame or any further packets; an endpoint
MAY send a Stateless Reset (Section 10.3) in response to further
packets that it receives.

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

Endpoints that track all individual packets for the purposes of
detecting duplicates are at risk of accumulating excessive state.
The data required for detecting duplicates can be limited by
maintaining a minimum packet number below which all packets are
immediately dropped. Any minimum needs to account for large
variations in round trip time, which includes the possibility that a
peer might probe network paths with much larger round trip times; see
Section 9.

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

12.4. Frames and Frame Types

The payload of QUIC packets, after removing packet protection,
consists of a sequence of complete frames, as shown in Figure 11.
Version Negotiation, Stateless Reset, and Retry packets do not
contain frames.

```
Packet Payload {
  Frame (8..) ...,
}
```

Figure 11: QUIC Payload

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

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

```
Frame {
  Frame Type (i),
  Type-Dependent Fields (..),
}
```

                      Figure 12: Generic Frame Layout

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

Type Value	Frame Type Name	Definition	Pkts	Spec
=============	=====================	================	======	======
0x00	PADDING	Section 19.1	IH01	NP
0x01	PING	Section 19.2	IH01	
0x02 - 0x03	ACK	Section 19.3	IH_1	NC
0x04	RESET_STREAM	Section 19.4	__01	
0x05	STOP_SENDING	Section 19.5	__01	
0x06	CRYPTO	Section 19.6	IH_1	
0x07	NEW_TOKEN	Section 19.7	___1	
0x08 - 0x0f	STREAM	Section 19.8	__01	F
0x10	MAX_DATA	Section 19.9	__01	
0x11	MAX_STREAM_DATA	Section 19.10	__01	
0x12 - 0x13	MAX_STREAMS	Section 19.11	__01	
0x14	DATA_BLOCKED	Section 19.12	__01	
0x15	STREAM_DATA_BLOCKED	Section 19.13	__01	
0x16 - 0x17	STREAMS_BLOCKED	Section 19.14	__01	
0x18	NEW_CONNECTION_ID	Section 19.15	__01	P
0x19	RETIRE_CONNECTION_ID	Section 19.16	__01	
0x1a	PATH_CHALLENGE	Section 19.17	__01	P
0x1b	PATH_RESPONSE	Section 19.18	__01	P
0x1c - 0x1d	CONNECTION_CLOSE	Section 19.19	ih01	N
0x1e	HANDSHAKE_DONE	Section 19.20	___1	

Table 3: Frame Types

The "Pkts" column in Table 3 lists the types of packets that each
frame type could appear in, indicated by the following characters:

I: Initial (Section 17.2.2)

H: Handshake (Section 17.2.4)

0: 0-RTT (Section 17.2.3)

1: 1-RTT (Section 17.3)

ih: Only a CONNECTION_CLOSE frame of type 0x1c can appear in Initial
    or Handshake packets.

For more detail about these restrictions, see Section 12.5. Note
that all frames can appear in 1-RTT packets. An endpoint MUST treat
receipt of a frame in a packet type that is not permitted as a
connection error of type PROTOCOL_VIOLATION.

The "Spec" column in Table 3 summarizes any special rules governing
the processing or generation of the frame type, as indicated by the
following characters:

N: Packets containing only frames with this marking are not ack-
    eliciting; see Section 13.2.

C: Packets containing only frames with this marking do not count
    toward bytes in flight for congestion control purposes; see
    [QUIC-RECOVERY].

P: Packets containing only frames with this marking can be used to
    probe new network paths during connection migration; see
    Section 9.1.

F: The content of frames with this marking are flow controlled; see
    Section 4.

The "Pkts" and "Spec" columns in Table 3 do not form part of the IANA
registry; see Section 22.3.

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

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

The Frame Type field uses a variable-length integer encoding (see
Section 16) with one exception. To ensure simple and efficient
implementations of frame parsing, a frame type MUST use the shortest
possible encoding. For frame types defined in this document, this

means a single-byte encoding, even though it is possible to encode
these values as a two-, four- or eight-byte variable-length integer.
For instance, though 0x4001 is a legitimate two-byte encoding for a
variable-length integer with a value of 1, PING frames are always
encoded as a single byte with the value 0x01. This rule applies to
all current and future QUIC frame types. An endpoint MAY treat the
receipt of a frame type that uses a longer encoding than necessary as
a connection error of type PROTOCOL_VIOLATION.

12.5. Frames and Number Spaces

Some frames are prohibited in different packet number spaces. The
rules here generalize those of TLS, in that frames associated with
establishing the connection can usually appear in packets in any
packet number space, whereas those associated with transferring data
can only appear in the application data packet number space:

* PADDING, PING, and CRYPTO frames MAY appear in any packet number
   space.

* CONNECTION_CLOSE frames signaling errors at the QUIC layer (type
   0x1c) MAY appear in any packet number space. CONNECTION_CLOSE
   frames signaling application errors (type 0x1d) MUST only appear
   in the application data packet number space.

* ACK frames MAY appear in any packet number space, but can only
   acknowledge packets that appeared in that packet number space.
   However, as noted below, 0-RTT packets cannot contain ACK frames.

* All other frame types MUST only be sent in the application data
   packet number space.

Note that it is not possible to send the following frames in 0-RTT
packets for various reasons: ACK, CRYPTO, HANDSHAKE_DONE, NEW_TOKEN,
PATH_RESPONSE, and RETIRE_CONNECTION_ID. A server MAY treat receipt
of these frames in 0-RTT packets as a connection error of type
PROTOCOL_VIOLATION.

13. Packetization and Reliability

A sender sends one or more frames in a QUIC packet; see Section 12.4.

A sender can minimize per-packet bandwidth and computational costs by
including as many frames as possible in each QUIC packet. A sender
MAY wait for a short period of time to collect multiple frames before
sending a packet that is not maximally packed, to avoid sending out
large numbers of small packets. An implementation MAY use knowledge
about application sending behavior or heuristics to determine whether

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

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

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

13.1. Packet Processing

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

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

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

13.2. Generating Acknowledgements

   Endpoints acknowledge all packets they receive and process. However,
   only ack-eliciting packets cause an ACK frame to be sent within the
   maximum ack delay. Packets that are not ack-eliciting are only
   acknowledged when an ACK frame is sent for other reasons.

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

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

13.2.1. Sending ACK Frames

Every packet SHOULD be acknowledged at least once, and ack-eliciting packets MUST be acknowledged at least once within the maximum delay an endpoint communicated using the max_ack_delay transport parameter; see Section 18.2. max_ack_delay declares an explicit contract: an endpoint promises to never intentionally delay acknowledgments of an ack-eliciting packet by more than the indicated value. If it does, any excess accrues to the RTT estimate and could result in spurious or delayed retransmissions from the peer. A sender uses the receiver's max_ack_delay value in determining timeouts for timer-based retransmission, as detailed in Section 6.2 of [QUIC-RECOVERY].

An endpoint MUST acknowledge all ack-eliciting Initial and Handshake packets immediately and all ack-eliciting 0-RTT and 1-RTT packets within its advertised max_ack_delay, with the following exception. Prior to handshake confirmation, an endpoint might not have packet protection keys for decrypting Handshake, 0-RTT, or 1-RTT packets when they are received. It might therefore buffer them and acknowledge them when the requisite keys become available.

Since packets containing only ACK frames are not congestion controlled, an endpoint MUST NOT send more than one such packet in response to receiving an ack-eliciting packet.

An endpoint MUST NOT send a non-ack-eliciting packet in response to a non-ack-eliciting packet, even if there are packet gaps that precede the received packet. This avoids an infinite feedback loop of acknowledgements, which could prevent the connection from ever becoming idle. Non-ack-eliciting packets are eventually acknowledged when the endpoint sends an ACK frame in response to other events.

In order to assist loss detection at the sender, an endpoint SHOULD generate and send an ACK frame without delay when it receives an ack-eliciting packet either:

*   when the received packet has a packet number less than another ack-eliciting packet that has been received, or

*   when the packet has a packet number larger than the highest-numbered ack-eliciting packet that has been received and there are missing packets between that packet and this packet.

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

The algorithms in [QUIC-RECOVERY] are expected to be resilient to receivers that do not follow the guidance offered above. However, an implementation should only deviate from these requirements after careful consideration of the performance implications of a change, for connections made by the endpoint and for other users of the network.

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

A receiver MUST NOT send an ack-eliciting frame in all packets that would otherwise be non-ack-eliciting, to avoid an infinite feedback loop of acknowledgements.

13.2.2. Acknowledgement Frequency

A receiver determines how frequently to send acknowledgements in response to ack-eliciting packets. This determination involves a trade-off.

Endpoints rely on timely acknowledgment to detect loss; see Section 6 of [QUIC-RECOVERY]. Window-based congestion controllers, such as the one in Section 7 of [QUIC-RECOVERY], rely on acknowledgments to manage their congestion window. In both cases, delaying acknowledgments can adversely affect performance.

On the other hand, reducing the frequency of packets that carry only acknowledgements reduces packet transmission and processing cost at both endpoints. It can improve connection throughput on severely asymmetric links and reduce the volume of acknowledgment traffic using return path capacity; see Section 3 of [RFC3449].

A receiver SHOULD send an ACK frame after receiving at least two ack-eliciting packets. This recommendation is general in nature and consistent with recommendations for TCP endpoint behavior [RFC5681]. Knowledge of network conditions, knowledge of the peer's congestion controller, or further research and experimentation might suggest alternative acknowledgment strategies with better performance characteristics.

A receiver MAY process multiple available packets before determining
whether to send an ACK frame in response.

13.2.3. Managing ACK Ranges

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

ACK frames SHOULD always acknowledge the most recently received
packets, and the more out-of-order the packets are, the more
important it is to send an updated ACK frame quickly, to prevent the
peer from declaring a packet as lost and spuriously retransmitting
the frames it contains. An ACK frame is expected to fit within a
single QUIC packet. If it does not, then older ranges (those with
the smallest packet numbers) are omitted.

A receiver limits the number of ACK Ranges (Section 19.3.1) it
remembers and sends in ACK frames, both to limit the size of ACK
frames and to avoid resource exhaustion. After receiving
acknowledgments for an ACK frame, the receiver SHOULD stop tracking
those acknowledged ACK Ranges. Senders can expect acknowledgements
for most packets, but QUIC does not guarantee receipt of an
acknowledgment for every packet that the receiver processes.

It is possible that retaining many ACK Ranges could cause an ACK
frame to become too large. A receiver can discard unacknowledged ACK
Ranges to limit ACK frame size, at the cost of increased
retransmissions from the sender. This is necessary if an ACK frame
would be too large to fit in a packet. Receivers MAY also limit ACK
frame size further to preserve space for other frames or to limit the
capacity that acknowledgments consume.

A receiver MUST retain an ACK Range unless it can ensure that it will
not subsequently accept packets with numbers in that range.
Maintaining a minimum packet number that increases as ranges are
discarded is one way to achieve this with minimal state.

Receivers can discard all ACK Ranges, but they MUST retain the
largest packet number that has been successfully processed as that is
used to recover packet numbers from subsequent packets; see
Section 17.1.

A receiver SHOULD include an ACK Range containing the largest
received packet number in every ACK frame. The Largest Acknowledged
field is used in ECN validation at a sender and including a lower
value than what was included in a previous ACK frame could cause ECN
to be unnecessarily disabled; see Section 13.4.2.

Section 13.2.4 describes an exemplary approach for determining what
packets to acknowledge in each ACK frame. Though the goal of this
algorithm is to generate an acknowledgment for every packet that is
processed, it is still possible for acknowledgments to be lost.

13.2.4. Limiting Ranges by Tracking ACK Frames

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

A receiver that sends only non-ack-eliciting packets, such as ACK
frames, might not receive an acknowledgement for a long period of
time. This could cause the receiver to maintain state for a large
number of ACK frames for a long period of time, and ACK frames it
sends could be unnecessarily large. In such a case, a receiver could
send a PING or other small ack-eliciting frame occasionally, such as
once per round trip, to elicit an ACK from the peer.

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

13.2.5. Measuring and Reporting Host Delay

An endpoint measures the delays intentionally introduced between the
time the packet with the largest packet number is received and the
time an acknowledgment is sent. The endpoint encodes this
acknowledgement delay in the ACK Delay field of an ACK frame; see
Section 19.3. This allows the receiver of the ACK frame to adjust
for any intentional delays, which is important for getting a better
estimate of the path RTT when acknowledgments are delayed.

A packet might be held in the OS kernel or elsewhere on the host
before being processed. An endpoint MUST NOT include delays that it
does not control when populating the ACK Delay field in an ACK frame.
However, endpoints SHOULD include buffering delays caused by
unavailability of decryption keys, since these delays can be large
and are likely to be non-repeating.

When the measured acknowledgement delay is larger than its
max_ack_delay, an endpoint SHOULD report the measured delay. This
information is especially useful during the handshake when delays
might be large; see Section 13.2.1.

13.2.6. ACK Frames and Packet Protection

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

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

13.2.7. PADDING Frames Consume Congestion Window

Packets containing PADDING frames are considered to be in flight for
congestion control purposes [QUIC-RECOVERY]. Packets containing only
PADDING frames therefore consume congestion window but do not
generate acknowledgments that will open the congestion window. To
avoid a deadlock, a sender SHOULD ensure that other frames are sent
periodically in addition to PADDING frames to elicit acknowledgments
from the receiver.

13.3. Retransmission of Information

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

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

*   Data sent in CRYPTO frames is retransmitted according to the rules
    in [QUIC-RECOVERY], until all data has been acknowledged. Data in
    CRYPTO frames for Initial and Handshake packets is discarded when
    keys for the corresponding packet number space are discarded.

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

*   ACK frames carry the most recent set of acknowledgements and the
    acknowledgement delay from the largest acknowledged packet, as
    described in Section 13.2.1. Delaying the transmission of packets
    containing ACK frames or resending old ACK frames can cause the
    peer to generate an inflated RTT sample or unnecessarily disable
    ECN.

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

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

*   Connection close signals, including packets that contain
    CONNECTION_CLOSE frames, are not sent again when packet loss is
    detected, but as described in Section 10.

*   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; see Section 4.2.

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

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

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

* 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. The peer is expected to send more PATH_CHALLENGE
  frames as necessary to evoke additional PATH_RESPONSE frames.

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

* NEW_TOKEN frames are retransmitted if the packet containing them
  is lost. No special support is made for detecting reordered and
  duplicated NEW_TOKEN frames other than a direct comparison of the
  frame contents.

* PING and PADDING frames contain no information, so lost PING or
  PADDING frames do not require repair.

* The HANDSHAKE_DONE frame MUST be retransmitted until it is
  acknowledged.

Endpoints SHOULD prioritize retransmission of data over sending new
data, unless priorities specified by the application indicate
otherwise; see Section 2.3.

Even though a sender is encouraged to assemble frames containing up-
to-date information every time it sends a packet, it is not forbidden
to retransmit copies of frames from lost packets. A sender that
retransmits copies of frames needs to handle decreases in available
payload size due to change in packet number length, connection ID
length, and path MTU. A receiver MUST accept packets containing an
outdated frame, such as a MAX_DATA frame carrying a smaller maximum
data than one found in an older packet.

A sender SHOULD avoid retransmitting information from packets once
they are acknowledged. This includes packets that are acknowledged
after being declared lost, which can happen in the presence of
network reordering. Doing so requires senders to retain information
about packets after they are declared lost. A sender can discard
this information after a period of time elapses that adequately
allows for reordering, such as a PTO (Section 6.2 of
[QUIC-RECOVERY]), or on other events, such as reaching a memory
limit.

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

13.4. Explicit Congestion Notification

QUIC endpoints can use Explicit Congestion Notification (ECN)
[RFC3168] to detect and respond to network congestion. ECN allows an
endpoint to set an ECT codepoint in the ECN field of an IP packet. A
network node can then indicate congestion by setting the CE codepoint
in the ECN field instead of dropping the packet [RFC8087]. Endpoints
react to reported congestion by reducing their sending rate in
response, as described in [QUIC-RECOVERY].

To enable ECN, a sending QUIC endpoint first determines whether a
path supports ECN marking and whether the peer reports the ECN values
in received IP headers; see Section 13.4.2.

13.4.1. Reporting ECN Counts

Use of ECN requires the receiving endpoint to read the ECN field from
an IP packet, which is not possible on all platforms. If an endpoint
does not implement ECN support or does not have access to received
ECN fields, it does not report ECN counts for packets it receives.

Even if an endpoint does not set an ECT field on packets it sends,
the endpoint MUST provide feedback about ECN markings it receives, if
these are accessible. Failing to report the ECN counts will cause
the sender to disable use of ECN for this connection.

On receiving an IP packet with an ECT(0), ECT(1) or CE codepoint, an ECN-enabled endpoint accesses the ECN field and increases the corresponding ECT(0), ECT(1), or CE count. These ECN counts are included in subsequent ACK frames; see Section 13.2 and Section 19.3.

Each packet number space maintains separate acknowledgement state and separate ECN counts. Coalesced QUIC packets (see Section 12.2) share the same IP header so the ECN counts are incremented once for each coalesced QUIC packet.

For example, if one each of an Initial, Handshake, and 1-RTT QUIC packet are coalesced into a single UDP datagram, the ECN counts for all three packet number spaces will be incremented by one each, based on the ECN field of the single IP header.

ECN counts are only incremented when QUIC packets from the received IP packet are processed. As such, duplicate QUIC packets are not processed and do not increase ECN counts; see Section 21.9 for relevant security concerns.

13.4.2. ECN Validation

It is possible for faulty network devices to corrupt or erroneously drop packets that carry a non-zero ECN codepoint. To ensure connectivity in the presence of such devices, an endpoint validates the ECN counts for each network path and disables use of ECN on that path if errors are detected.

To perform ECN validation for a new path:

* The endpoint sets an ECT(0) codepoint in the IP header of early outgoing packets sent on a new path to the peer ([RFC8311]).

* The endpoint monitors whether all packets sent with an ECT codepoint are eventually deemed lost (Section 6 of [QUIC-RECOVERY]), indicating that ECN validation has failed.

If an endpoint has cause to expect that IP packets with an ECT codepoint might be dropped by a faulty network element, the endpoint could set an ECT codepoint for only the first ten outgoing packets on a path, or for a period of three PTOs (see Section 6.2 of [QUIC-RECOVERY]). If all packets marked with non-zero ECN codepoints are subsequently lost, it can disable marking on the assumption that the marking caused the loss.

An endpoint thus attempts to use ECN and validates this for each new
connection, when switching to a server's preferred address, and on
active connection migration to a new path. Appendix B describes one
possible algorithm.

Other methods of probing paths for ECN support are possible, as are
different marking strategies. Implementations MAY use other methods
defined in RFCs; see [RFC8311]. Implementations that use the ECT(1)
codepoint need to perform ECN validation using the reported ECT(1)
counts.

13.4.2.1. Receiving ACK Frames with ECN Counts

Erroneous application of CE markings by the network can result in
degraded connection performance. An endpoint that receives an ACK
frame with ECN counts therefore validates the counts before using
them. It performs this validation by comparing newly received counts
against those from the last successfully processed ACK frame. Any
increase in the ECN counts is validated based on the ECN markings
that were applied to packets that are newly acknowledged in the ACK
frame.

If an ACK frame newly acknowledges a packet that the endpoint sent
with either the ECT(0) or ECT(1) codepoint set, ECN validation fails
if the corresponding ECN counts are not present in the ACK frame.
This check detects a network element that zeroes the ECN field or a
peer that does not report ECN markings.

ECN validation also fails if the sum of the increase in ECT(0) and
ECN-CE counts is less than the number of newly acknowledged packets
that were originally sent with an ECT(0) marking. Similarly, ECN
validation fails if the sum of the increases to ECT(1) and ECN-CE
counts is less than the number of newly acknowledged packets sent
with an ECT(1) marking. These checks can detect remarking of ECN-CE
markings by the network.

An endpoint could miss acknowledgements for a packet when ACK frames
are lost. It is therefore possible for the total increase in ECT(0),
ECT(1), and ECN-CE counts to be greater than the number of packets
that are newly acknowledged by an ACK frame. This is why ECN counts
are permitted to be larger than the total number of packets that are
acknowledged.

Validating ECN counts from reordered ACK frames can result in
failure. An endpoint MUST NOT fail ECN validation as a result of
processing an ACK frame that does not increase the largest
acknowledged packet number.

ECN validation can fail if the received total count for either ECT(0) or ECT(1) exceeds the total number of packets sent with each corresponding ECT codepoint. In particular, validation will fail when an endpoint receives a non-zero ECN count corresponding to an ECT codepoint that it never applied. This check detects when packets are remarked to ECT(0) or ECT(1) in the network.

13.4.2.2. ECN Validation Outcomes

If validation fails, then the endpoint MUST disable ECN. It stops setting the ECT codepoint in IP packets that it sends, assuming that either the network path or the peer does not support ECN.

Even if validation fails, an endpoint MAY revalidate ECN for the same path at any later time in the connection. An endpoint could continue to periodically attempt validation.

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

14. Datagram Size

A UDP datagram can include one or more QUIC packets. The datagram size refers to the total UDP payload size of a single UDP datagram carrying QUIC packets. The datagram size includes one or more QUIC packet headers and protected payloads, but not the UDP or IP headers.

The maximum datagram size is defined as the largest size of UDP payload that can be sent across a network path using a single UDP datagram.

QUIC depends upon a minimum IP packet size of at least 1280 bytes. This is the IPv6 minimum size ([IPv6]) and is also supported by most modern IPv4 networks. Assuming the minimum IP header size of 40 bytes for IPv6 and 20 bytes for IPv4 and a UDP header size of 8 bytes, this results in a maximum datagram size of 1232 bytes for IPv6 and 1252 bytes for IPv4.

The maximum datagram size MUST be at least 1200 bytes.

Any maximum datagram size larger than 1200 bytes can be discovered using Path Maximum Transmission Unit Discovery (PMTUD; see Section 14.2.1) or Datagram Packetization Layer PMTU Discovery (DPLPMTUD; see Section 14.3).

Enforcement of the max_udp_payload_size transport parameter
(Section 18.2) might act as an additional limit on the maximum
datagram size. A sender can avoid exceeding this limit, once the
value is known. However, prior to learning the value of the
transport parameter, endpoints risk datagrams being lost if they send
datagrams larger than the smallest allowed maximum datagram size of
1200 bytes.

UDP datagrams MUST NOT be fragmented at the IP layer. In IPv4
([IPv4]), the DF bit MUST be set if possible, to prevent
fragmentation on the path.

Datagrams are required to be of a minimum size under some conditions.
However, the size of the datagram is not authenticated. Therefore,
an endpoint MUST NOT close a connection when it receives a datagram
that does not meet size constraints, though the endpoint MAY discard
such datagrams.

14.1. Initial Datagram Size

A client MUST expand the payload of all UDP datagrams carrying
Initial packets to at least the smallest allowed maximum datagram
size of 1200 bytes by adding PADDING frames to the Initial packet or
by coalescing the Initial packet; see Section 12.2. Similarly, a
server MUST expand the payload of all UDP datagrams carrying ack-
eliciting Initial packets to at least the smallest allowed maximum
datagram size of 1200 bytes. Sending UDP datagrams of this size
ensures that the network path supports a reasonable Path Maximum
Transmission Unit (PMTU), in both directions. Additionally, a client
that expands Initial packets helps reduce the amplitude of
amplification attacks caused by server responses toward an unverified
client address; see Section 8.

Datagrams containing Initial packets MAY exceed 1200 bytes if the
sender believes that the network path and peer both support the size
that it chooses.

A server MUST discard an Initial packet that is carried in a UDP
datagram with a payload that is smaller than the smallest allowed
maximum datagram size of 1200 bytes. A server MAY also immediately
close the connection by sending a CONNECTION_CLOSE frame with an
error code of PROTOCOL_VIOLATION; see Section 10.2.3.

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

14.2. Path Maximum Transmission Unit

   The Path Maximum Transmission Unit (PMTU) is the maximum size of the
   entire IP packet including the IP header, UDP header, and UDP
   payload. The UDP payload includes one or more QUIC packet headers
   and protected payloads. The PMTU can depend on path characteristics,
   and can therefore change over time. The largest UDP payload an
   endpoint sends at any given time is referred to as the endpoint's
   maximum datagram size.

   An endpoint SHOULD use DPLPMTUD (Section 14.3) or PMTUD
   (Section 14.2.1) to determine whether the path to a destination will
   support a desired maximum datagram size without fragmentation. In
   the absence of these mechanisms, QUIC endpoints SHOULD NOT send
   datagrams larger than the smallest allowed maximum datagram size.

   Both DPLPMTUD and PMTUD send datagrams that are larger than the
   current maximum datagram size, referred to as PMTU probes. All QUIC
   packets that are not sent in a PMTU probe SHOULD be sized to fit
   within the maximum datagram size to avoid the datagram being
   fragmented or dropped ([RFC8085]).

   If a QUIC endpoint determines that the PMTU between any pair of local
   and remote IP addresses has fallen below the smallest allowed maximum
   datagram size of 1200 bytes, it MUST immediately cease sending QUIC
   packets, except for those in PMTU probes or those containing
   CONNECTION_CLOSE frames, on the affected path. An endpoint MAY
   terminate the connection if an alternative path cannot be found.

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

   A QUIC implementation MAY be more conservative in computing the
   maximum datagram size to allow for unknown tunnel overheads or IP
   header options/extensions.

14.2.1. Handling of ICMP Messages by PMTUD

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

   An endpoint MUST ignore an ICMP message that claims the PMTU has
   decreased below QUIC's smallest allowed maximum datagram size.

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

   QUIC endpoints using PMTUD SHOULD validate ICMP messages to protect
   from off-path injection as specified in [RFC8201] and Section 5.2 of
   [RFC8085]. This validation SHOULD use the quoted packet supplied in
   the payload of an ICMP message to associate the message with a
   corresponding transport connection (see Section 4.6.1 of [DPLPMTUD]).
   ICMP message validation MUST include matching IP addresses and UDP
   ports ([RFC8085]) and, when possible, connection IDs to an active
   QUIC session. The endpoint SHOULD ignore all ICMP messages that fail
   validation.

   An endpoint MUST NOT increase PMTU based on ICMP messages; see
   Section 3, clause 6 of [DPLPMTUD]. Any reduction in QUIC's maximum
   datagram size in response to ICMP messages MAY be provisional until
   QUIC's loss detection algorithm determines that the quoted packet has
   actually been lost.

14.3. Datagram Packetization Layer PMTU Discovery

   Datagram Packetization Layer PMTU Discovery (DPLPMTUD; [DPLPMTUD])
   relies on tracking loss or acknowledgment of QUIC packets that are
   carried in PMTU probes. PMTU probes for DPLPMTUD that use the
   PADDING frame implement "Probing using padding data", as defined in
   Section 4.1 of [DPLPMTUD].

   Endpoints SHOULD set the initial value of BASE_PLPMTU (Section 5.1 of
   [DPLPMTUD]) to be consistent with QUIC's smallest allowed maximum
   datagram size. The MIN_PLPMTU is the same as the BASE_PLPMTU.

   QUIC endpoints implementing DPLPMTUD maintain a DPLPMTUD Maximum
   Packet Size (MPS, Section 4.4 of [DPLPMTUD]) for each combination of
   local and remote IP addresses. This corresponds to the maximum
   datagram size.

14.3.1. DPLPMTUD and Initial Connectivity

From the perspective of DPLPMTUD, QUIC is an acknowledged
Packetization Layer (PL). A QUIC sender can therefore enter the
DPLPMTUD BASE state (Section 5.2 of [DPLPMTUD]) when the QUIC
connection handshake has been completed.

14.3.2. Validating the Network Path with DPLPMTUD

QUIC is an acknowledged PL, therefore a QUIC sender does not
implement a DPLPMTUD CONFIRMATION_TIMER while in the SEARCH_COMPLETE
state; see Section 5.2 of [DPLPMTUD].

14.3.3. Handling of ICMP Messages by DPLPMTUD

An endpoint using DPLPMTUD requires the validation of any received
ICMP Packet Too Big (PTB) message before using the PTB information,
as defined in Section 4.6 of [DPLPMTUD]. In addition to UDP port
validation, QUIC validates an ICMP message by using other PL
information (e.g., validation of connection IDs in the quoted packet
of any received ICMP message).

The considerations for processing ICMP messages described in
Section 14.2.1 also apply if these messages are used by DPLPMTUD.

14.4. Sending QUIC PMTU Probes

PMTU probes are ack-eliciting packets.

Endpoints could limit the content of PMTU probes to PING and PADDING
frames, since packets that are larger than the current maximum
datagram size are more likely to be dropped by the network. Loss of
a QUIC packet that is carried in a PMTU probe is therefore not a
reliable indication of congestion and SHOULD NOT trigger a congestion
control reaction; see Section 3, Bullet 7 of [DPLPMTUD]. However,
PMTU probes consume congestion window, which could delay subsequent
transmission by an application.

14.4.1. PMTU Probes Containing Source Connection ID

Endpoints that rely on the destination connection ID for routing
incoming QUIC packets are likely to require that the connection ID be
included in PMTU probes to route any resulting ICMP messages
(Section 14.2.1) back to the correct endpoint. However, only long
header packets (Section 17.2) contain the Source Connection ID field,
and long header packets are not decrypted or acknowledged by the peer
once the handshake is complete.

One way to construct a PMTU probe is to coalesce (see Section 12.2) a
packet with a long header, such as a Handshake or 0-RTT packet
(Section 17.2), with a short header packet in a single UDP datagram.
If the resulting PMTU probe reaches the endpoint, the packet with the
long header will be ignored, but the short header packet will be
acknowledged. If the PMTU probe causes an ICMP message to be sent,
the first part of the probe will be quoted in that message. If the
Source Connection ID field is within the quoted portion of the probe,
that could be used for routing or validation of the ICMP message.

Note: The purpose of using a packet with a long header is only to
   ensure that the quoted packet contained in the ICMP message
   contains a Source Connection ID field. This packet does not need
   to be a valid packet and it can be sent even if there is no
   current use for packets of that type.

15. Versions

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

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

Other versions of QUIC might have different properties from 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 bytes is 1010 (in binary). A
client or server MAY advertise support for any of these reserved
versions.

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

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

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

Version numbers used to identify IETF drafts are created by adding
the draft number to 0xff000000. For example, draft-ietf-quic-
transport-13 would be identified as 0xff00000d.

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

16. Variable-Length Integer Encoding

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

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

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

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

Table 4: Summary of Integer Encodings

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

Versions (Section 15) and packet numbers sent in the header
(Section 17.1) are described using integers, but do not use this
encoding.

17. Packet Formats

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

17.1. Packet Number Encoding and Decoding

Packet numbers are integers in the range 0 to $2^{62}-1$ (Section 12.3).
When present in long or short packet headers, they are encoded in 1
to 4 bytes. The number of bits required to represent the packet
number is reduced by including only the least significant bits of the
packet number.

The encoded packet number is protected as described in Section 5.4 of
[QUIC-TLS].

Prior to receiving an acknowledgement for a packet number space, the
full packet number MUST be included; it is not to be truncated as
described below.

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

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

   For example, if an endpoint has received an acknowledgment for packet
   0xabe8bc, sending a packet with a number of 0xac5c02 requires a
   packet number encoding with 16 bits or more; whereas the 24-bit
   packet number encoding is needed to send a packet with a number of
   0xace8fe.

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

   Once header protection is removed, the packet number is decoded by
   finding the packet number value that is closest to the next expected
   packet. The next expected packet is the highest received packet
   number plus one. For example, if the highest successfully
   authenticated packet had a packet number of 0xa82f30ea, then a packet
   containing a 16-bit value of 0x9b32 will be decoded as 0xa82f9b32.
   Example pseudo-code for packet number decoding can be found in
   Appendix A.

17.2. Long Header Packets

   Long Header Packet {
     Header Form (1) = 1,
     Fixed Bit (1) = 1,
     Long Packet Type (2),
     Type-Specific Bits (4),
     Version (32),
     Destination Connection ID Length (8),
     Destination Connection ID (0..160),
     Source Connection ID Length (8),
     Source Connection ID (0..160),
   }

                  Figure 13: Long Header Packet Format

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

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

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

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

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

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

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

Destination Connection ID: The Destination Connection ID field
   follows the Destination Connection ID Length field, which
   indicates the length of this field. Section 7.2 describes the use
   of this field in more detail.

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

Source Connection ID: The Source Connection ID field follows the
   Source Connection ID Length field, which indicates the length of
   this field. Section 7.2 describes the use of this field in more
   detail.

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

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

Table 5: Long Header Packet Types

The header form bit, Destination and Source Connection ID lengths,
Destination and Source Connection ID fields, and Version fields of a
long header packet are version-independent. The other fields in the
first byte are version-specific. See [QUIC-INVARIANTS] for details
on how packets from different versions of QUIC are interpreted.

The interpretation of the fields and the payload are specific to a
version and packet type. While type-specific semantics for this
version are described in the following sections, several long-header
packets in this version of QUIC contain these additional fields:

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

Packet Number Length: In packet types that contain a Packet Number
   field, the least significant two bits (those with a mask of 0x03)
   of byte 0 contain the length of the packet number, encoded as an
   unsigned, two-bit integer that is one less than the length of the
   packet number field in bytes. That is, the length of the packet
   number field is the value of this field, plus one. These bits are
   protected using header protection; see Section 5.4 of [QUIC-TLS].

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

Packet Number: The packet number field is 1 to 4 bytes long. The

packet number is protected using header protection; see
Section 5.4 of [QUIC-TLS]. The length of the packet number field
is encoded in the Packet Number Length bits of byte 0; see above.

17.2.1. Version Negotiation Packet

A Version Negotiation packet is inherently not version-specific.
Upon receipt by a client, it will be identified as a Version
Negotiation packet based on the Version field having a value of 0.

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

The layout of a Version Negotiation packet is:

```
Version Negotiation Packet {
  Header Form (1) = 1,
  Unused (7),
  Version (32) = 0,
  Destination Connection ID Length (8),
  Destination Connection ID (0..2040),
  Source Connection ID Length (8),
  Source Connection ID (0..2040),
  Supported Version (32) ...,
}
```

Figure 14: Version Negotiation Packet

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

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

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

Future versions of QUIC may have different requirements for the
lengths of connection IDs. In particular, connection IDs might have
a smaller minimum length or a greater maximum length. Version-
specific rules for the connection ID therefore MUST NOT influence a
server decision about whether to send a Version Negotiation packet.

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

A Version Negotiation packet is not acknowledged. It is only sent in
response to a packet that indicates an unsupported version; see
Section 5.2.2.

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

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

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

17.2.2. Initial Packet

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

```
Initial Packet {
  Header Form (1) = 1,
  Fixed Bit (1) = 1,
  Long Packet Type (2) = 0,
  Reserved Bits (2),
  Packet Number Length (2),
  Version (32),
  Destination Connection ID Length (8),
  Destination Connection ID (0..160),
  Source Connection ID Length (8),
  Source Connection ID (0..160),
  Token Length (i),
  Token (..),
  Length (i),
  Packet Number (8..32),
  Packet Payload (..),
}
```

                    Figure 15: Initial Packet

The Initial packet contains a long header as well as the Length and
Packet Number fields; see Section 17.2. The first byte contains the
Reserved and Packet Number Length bits; see also Section 17.2.
Between the Source Connection ID and Length fields, there are two
additional fields specific to the Initial packet.

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

Token: The value of the token that was previously provided in a
   Retry packet or NEW_TOKEN frame; see Section 8.1.

Packet Payload: The payload of the packet.

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

The client and server use the Initial packet type for any packet that
contains an initial cryptographic handshake message. This includes
all cases where a new packet containing the initial cryptographic
message needs to be created, such as the packets sent after receiving
a Retry packet (Section 17.2.5).

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

The payload of an Initial packet includes a CRYPTO frame (or frames)
containing a cryptographic handshake message, ACK frames, or both.
PING, PADDING, and CONNECTION_CLOSE frames of type 0x1c are also
permitted. An endpoint that receives an Initial packet containing
other frames can either discard the packet as spurious or treat it as
a connection error.

The first packet sent by a client always includes a CRYPTO frame that
contains the start or all of the first cryptographic handshake
message. The first CRYPTO frame sent always begins at an offset of
0; see Section 7.

Note that if the server sends a HelloRetryRequest, the client will
send another series of Initial packets. These Initial packets will
continue the cryptographic handshake and will contain CRYPTO frames
starting at an offset matching the size of the CRYPTO frames sent in
the first flight of Initial packets.

17.2.2.1. Abandoning Initial Packets

A client stops both sending and processing Initial packets when it
sends its first Handshake packet. A server stops sending and
processing Initial packets when it receives its first Handshake
packet. Though packets might still be in flight or awaiting
acknowledgment, no further Initial packets need to be exchanged
beyond this point. Initial packet protection keys are discarded (see
Section 4.9.1 of [QUIC-TLS]) along with any loss recovery and
congestion control state; see Section 6.4 of [QUIC-RECOVERY].

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

17.2.3. 0-RTT

A 0-RTT packet uses long headers with a type value of 0x1, followed
by the Length and Packet Number fields; see Section 17.2. The first
byte contains the Reserved and Packet Number Length bits; see
Section 17.2. A 0-RTT packet is used to carry "early" data from the
client to the server as part of the first flight, prior to handshake
completion. As part of the TLS handshake, the server can accept or
reject this early data.

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

```
0-RTT Packet {
  Header Form (1) = 1,
  Fixed Bit (1) = 1,
  Long Packet Type (2) = 1,
  Reserved Bits (2),
  Packet Number Length (2),
  Version (32),
  Destination Connection ID Length (8),
  Destination Connection ID (0..160),
  Source Connection ID Length (8),
  Source Connection ID (0..160),
  Length (i),
  Packet Number (8..32),
  Packet Payload (..),
}
```

                        Figure 16: 0-RTT Packet

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

   After a client receives a Retry packet, 0-RTT packets are likely to
   have been lost or discarded by the server. A client SHOULD attempt
   to resend data in 0-RTT packets after it sends a new Initial packet.
   New packet numbers MUST be used for any new packets that are sent; as
   described in Section 17.2.5.3, reusing packet numbers could
   compromise packet protection.

   A client only receives acknowledgments for its 0-RTT packets once the
   handshake is complete, as defined Section 4.1.1 of [QUIC-TLS].

   A client MUST NOT send 0-RTT packets once it starts processing 1-RTT
   packets from the server. This means that 0-RTT packets cannot
   contain any response to frames from 1-RTT packets. For instance, a
   client cannot send an ACK frame in a 0-RTT packet, because that can
   only acknowledge a 1-RTT packet. An acknowledgment for a 1-RTT
   packet MUST be carried in a 1-RTT packet.

   A server SHOULD treat a violation of remembered limits
   (Section 7.4.1) as a connection error of an appropriate type (for
   instance, a FLOW_CONTROL_ERROR for exceeding stream data limits).

17.2.4. Handshake Packet

   A Handshake packet uses long headers with a type value of 0x2,
   followed by the Length and Packet Number fields; see Section 17.2.
   The first byte contains the Reserved and Packet Number Length bits;
   see Section 17.2. It is used to carry cryptographic handshake
   messages and acknowledgments from the server and client.

```
Handshake Packet {
  Header Form (1) = 1,
  Fixed Bit (1) = 1,
  Long Packet Type (2) = 2,
  Reserved Bits (2),
  Packet Number Length (2),
  Version (32),
  Destination Connection ID Length (8),
  Destination Connection ID (0..160),
  Source Connection ID Length (8),
  Source Connection ID (0..160),
  Length (i),
  Packet Number (8..32),
  Packet Payload (..),
}
```

                  Figure 17: Handshake Protected Packet

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

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

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

   The payload of this packet contains CRYPTO frames and could contain
   PING, PADDING, or ACK frames. Handshake packets MAY contain
   CONNECTION_CLOSE frames of type 0x1c. Endpoints MUST treat receipt
   of Handshake packets with other frames as a connection error.

   Like Initial packets (see Section 17.2.2.1), data in CRYPTO frames
   for Handshake packets is discarded - and no longer retransmitted -
   when Handshake protection keys are discarded.

17.2.5. Retry Packet

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

```
Retry Packet {
  Header Form (1) = 1,
  Fixed Bit (1) = 1,
  Long Packet Type (2) = 3,
  Unused (4),
  Version (32),
  Destination Connection ID Length (8),
  Destination Connection ID (0..160),
  Source Connection ID Length (8),
  Source Connection ID (0..160),
  Retry Token (..),
  Retry Integrity Tag (128),
}
```

Figure 18: Retry Packet

A Retry packet (shown in Figure 18) does not contain any protected
fields. The value in the Unused field is set to an arbitrary value
by the server; a client MUST ignore these bits. In addition to the
fields from the long header, it contains these additional fields:

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

Retry Integrity Tag: See the Retry Packet Integrity section of
   [QUIC-TLS].

17.2.5.1. Sending a Retry Packet

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

The server includes a connection ID of its choice in the Source
Connection ID field. This value MUST NOT be equal to the Destination
Connection ID field of the packet sent by the client. A client MUST
discard a Retry packet that contains a Source Connection ID field
that is identical to the Destination Connection ID field of its
Initial packet. The client MUST use the value from the Source
Connection ID field of the Retry packet in the Destination Connection
ID field of subsequent packets that it sends.

A server MAY send Retry packets in response to Initial and 0-RTT
packets. A server can either discard or buffer 0-RTT packets that it
receives. A server can send multiple Retry packets as it receives
Initial or 0-RTT packets. A server MUST NOT send more than one Retry
packet in response to a single UDP datagram.

17.2.5.2. Handling a Retry Packet

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

   Clients MUST discard Retry packets that have a Retry Integrity Tag
   that cannot be validated; see the Retry Packet Integrity section of
   [QUIC-TLS]. This diminishes an off-path attacker's ability to inject
   a Retry packet and protects against accidental corruption of Retry
   packets. A client MUST discard a Retry packet with a zero-length
   Retry Token field.

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

   A client sets the Destination Connection ID field of this Initial
   packet to the value from the Source Connection ID in the Retry
   packet. Changing Destination Connection ID also results in a change
   to the keys used to protect the Initial packet. It also sets the
   Token field to the token provided in the Retry. The client MUST NOT
   change the Source Connection ID because the server could include the
   connection ID as part of its token validation logic; see
   Section 8.1.4.

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

17.2.5.3. Continuing a Handshake After Retry

   Subsequent Initial packets from the client include the connection ID
   and token values from the Retry packet. The client copies the Source
   Connection ID field from the Retry packet to the Destination
   Connection ID field and uses this value until an Initial packet with
   an updated value is received; see Section 7.2. The value of the
   Token field is copied to all subsequent Initial packets; see
   Section 8.1.2.

   Other than updating the Destination Connection ID and Token fields,
   the Initial packet sent by the client is subject to the same
   restrictions as the first Initial packet. A client MUST use the same
   cryptographic handshake message it included in this packet. A server
   MAY treat a packet that contains a different cryptographic handshake
   message as a connection error or discard it.

A client MAY attempt 0-RTT after receiving a Retry packet by sending
0-RTT packets to the connection ID provided by the server. A client
MUST NOT change the cryptographic handshake message it sends in
response to receiving a Retry.

A client MUST NOT reset the packet number for any packet number space
after processing a Retry packet. In particular, 0-RTT packets
contain confidential information that will most likely be
retransmitted on receiving a Retry packet. The keys used to protect
these new 0-RTT packets will not change as a result of responding to
a Retry packet. However, the data sent in these packets could be
different than what was sent earlier. Sending these new packets with
the same packet number is likely to compromise the packet protection
for those packets because the same key and nonce could be used to
protect different content. A server MAY abort the connection if it
detects that the client reset the packet number.

The connection IDs used on Initial and Retry packets exchanged
between client and server are copied to the transport parameters and
validated as described in Section 7.3.

17.3. Short Header Packets

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

```
Short Header Packet {
  Header Form (1) = 0,
  Fixed Bit (1) = 1,
  Spin Bit (1),
  Reserved Bits (2),
  Key Phase (1),
  Packet Number Length (2),
  Destination Connection ID (0..160),
  Packet Number (8..32),
  Packet Payload (..),
}
```

Figure 19: Short Header Packet Format

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

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

Fixed Bit: The next bit (0x40) of byte 0 is set to 1. Packets

containing a zero value for this bit are not valid packets in this
version and MUST be discarded.

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

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

Key Phase: The next bit (0x04) of byte 0 indicates the key phase,
   which allows a recipient of a packet to identify the packet
   protection keys that are used to protect the packet. See
   [QUIC-TLS] for details. This bit is protected using header
   protection; see Section 5.4 of [QUIC-TLS].

Packet Number Length: The least significant two bits (those with a
   mask of 0x03) of byte 0 contain the length of the packet number,
   encoded as an unsigned, two-bit integer that is one less than the
   length of the packet number field in bytes. That is, the length
   of the packet number field is the value of this field, plus one.
   These bits are protected using header protection; see Section 5.4
   of [QUIC-TLS].

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

Packet Number: The packet number field is 1 to 4 bytes long. The
   packet number has confidentiality protection separate from packet
   protection, as described in Section 5.4 of [QUIC-TLS]. The length
   of the packet number field is encoded in Packet Number Length
   field. See Section 17.1 for details.

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

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

17.3.1. Latency Spin Bit

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

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

   Each endpoint unilaterally decides if the spin bit is enabled or
   disabled for a connection. Implementations MUST allow administrators
   of clients and servers to disable the spin bit either globally or on
   a per-connection basis. Even when the spin bit is not disabled by
   the administrator, endpoints MUST disable their use of the spin bit
   for a random selection of at least one in every 16 network paths, or
   for one in every 16 connection IDs. As each endpoint disables the
   spin bit independently, this ensures that the spin bit signal is
   disabled on approximately one in eight network paths.

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

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

   When a server receives a short header packet that increases the
   highest packet number seen by the server from the client on a given
   network path, it sets the spin value for that path to be equal to the
   spin bit in the received packet.

   When a client receives a short header packet that increases the
   highest packet number seen by the client from the server on a given
   network path, it sets the spin value for that path to the inverse of
   the spin bit in the received packet.

   An endpoint resets the spin value for a network path to zero when
   changing the connection ID being used on that network path.

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

18. Transport Parameter Encoding

   The extension_data field of the quic_transport_parameters extension
   defined in [QUIC-TLS] contains the QUIC transport parameters. They
   are encoded as a sequence of transport parameters, as shown in
   Figure 20:

   Transport Parameters {
     Transport Parameter (..) ...,
   }

                 Figure 20: Sequence of Transport Parameters

   Each transport parameter is encoded as an (identifier, length, value)
   tuple, as shown in Figure 21:

   Transport Parameter {
     Transport Parameter ID (i),
     Transport Parameter Length (i),
     Transport Parameter Value (..),
   }

                   Figure 21: Transport Parameter Encoding

   The Transport Parameter Length field contains the length of the
   Transport Parameter Value field.

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

18.1. Reserved Transport Parameters

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

18.2. Transport Parameter Definitions

   This section details the transport parameters defined in this
   document.

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

   The following transport parameters are defined:

   original_destination_connection_id (0x00): The value of the
      Destination Connection ID field from the first Initial packet sent
      by the client; see Section 7.3. This transport parameter is only
      sent by a server.

   max_idle_timeout (0x01): The max idle timeout is a value in
      milliseconds that is encoded as an integer; see (Section 10.1).
      Idle timeout is disabled when both endpoints omit this transport
      parameter or specify a value of 0.

   stateless_reset_token (0x02): A stateless reset token is used in
      verifying a stateless reset; see Section 10.3. This parameter is
      a sequence of 16 bytes. This transport parameter MUST NOT be sent
      by a client, but MAY be sent by a server. A server that does not
      send this transport parameter cannot use stateless reset
      (Section 10.3) for the connection ID negotiated during the
      handshake.

   max_udp_payload_size (0x03): The maximum UDP payload size parameter
      is an integer value that limits the size of UDP payloads that the
      endpoint is willing to receive. UDP datagrams with payloads
      larger than this limit are not likely to be processed by the
      receiver.

      The default for this parameter is the maximum permitted UDP
      payload of 65527. Values below 1200 are invalid.

      This limit does act as an additional constraint on datagram size
      in the same way as the path MTU, but it is a property of the
      endpoint and not the path; see Section 14. It is expected that
      this is the space an endpoint dedicates to holding incoming
      packets.

   initial_max_data (0x04): The initial maximum data parameter is an

integer value that contains the initial value for the maximum
amount of data that can be sent on the connection. This is
equivalent to sending a MAX_DATA (Section 19.9) for the connection
immediately after completing the handshake.

initial_max_stream_data_bidi_local (0x05): This parameter is an
integer value specifying the initial flow control limit for
locally-initiated bidirectional streams. This limit applies to
newly created bidirectional streams opened by the endpoint that
sends the transport parameter. In client transport parameters,
this applies to streams with an identifier with the least
significant two bits set to 0x0; in server transport parameters,
this applies to streams with the least significant two bits set to
0x1.

initial_max_stream_data_bidi_remote (0x06): This parameter is an
integer value specifying the initial flow control limit for peer-
initiated bidirectional streams. This limit applies to newly
created bidirectional streams opened by the endpoint that receives
the transport parameter. In client transport parameters, this
applies to streams with an identifier with the least significant
two bits set to 0x1; in server transport parameters, this applies
to streams with the least significant two bits set to 0x0.

initial_max_stream_data_uni (0x07): This parameter is an integer
value specifying the initial flow control limit for unidirectional
streams. This limit applies to newly created unidirectional
streams opened by the endpoint that receives the transport
parameter. In client transport parameters, this applies to
streams with an identifier with the least significant two bits set
to 0x3; in server transport parameters, this applies to streams
with the least significant two bits set to 0x2.

initial_max_streams_bidi (0x08): The initial maximum bidirectional
streams parameter is an integer value that contains the initial
maximum number of bidirectional streams the peer may initiate. If
this parameter is absent or zero, the peer cannot open
bidirectional streams until a MAX_STREAMS frame is sent. Setting
this parameter is equivalent to sending a MAX_STREAMS
(Section 19.11) of the corresponding type with the same value.

initial_max_streams_uni (0x09): The initial maximum unidirectional
streams parameter is an integer value that contains the initial
maximum number of unidirectional streams the peer may initiate.
If this parameter is absent or zero, the peer cannot open
unidirectional streams until a MAX_STREAMS frame is sent. Setting
this parameter is equivalent to sending a MAX_STREAMS
(Section 19.11) of the corresponding type with the same value.

ack_delay_exponent (0x0a): The acknowledgement delay exponent is an
   integer value indicating an exponent used to decode the ACK Delay
   field in the ACK frame (Section 19.3). If this value is absent, a
   default value of 3 is assumed (indicating a multiplier of 8).
   Values above 20 are invalid.

max_ack_delay (0x0b): The maximum acknowledgement delay is an
   integer value indicating the maximum amount of time in
   milliseconds by which the endpoint will delay sending
   acknowledgments. This value SHOULD include the receiver's
   expected delays in alarms firing. For example, if a receiver sets
   a timer for 5ms and alarms commonly fire up to 1ms late, then it
   should send a max_ack_delay of 6ms. If this value is absent, a
   default of 25 milliseconds is assumed. Values of 2^14 or greater
   are invalid.

disable_active_migration (0x0c): The disable active migration
   transport parameter is included if the endpoint does not support
   active connection migration (Section 9) on the address being used
   during the handshake. When a peer sets this transport parameter,
   an endpoint MUST NOT use a new local address when sending to the
   address that the peer used during the handshake. This transport
   parameter does not prohibit connection migration after a client
   has acted on a preferred_address transport parameter. This
   parameter is a zero-length value.

preferred_address (0x0d): The server's preferred address is used to
   effect a change in server address at the end of the handshake, as
   described in Section 9.6. This transport parameter is only sent
   by a server. Servers MAY choose to only send a preferred address
   of one address family by sending an all-zero address and port
   (0.0.0.0:0 or ::.0) for the other family. IP addresses are
   encoded in network byte order.

   The preferred_address transport parameter contains an address and
   port for both IP version 4 and 6. The four-byte IPv4 Address
   field is followed by the associated two-byte IPv4 Port field.
   This is followed by a 16-byte IPv6 Address field and two-byte IPv6
   Port field. After address and port pairs, a Connection ID Length
   field describes the length of the following Connection ID field.
   Finally, a 16-byte Stateless Reset Token field includes the
   stateless reset token associated with the connection ID. The
   format of this transport parameter is shown in Figure 22.

   The Connection ID field and the Stateless Reset Token field

contain an alternative connection ID that has a sequence number of
1; see Section 5.1.1. Having these values sent alongside the
preferred address ensures that there will be at least one unused
active connection ID when the client initiates migration to the
preferred address.

The Connection ID and Stateless Reset Token fields of a preferred
address are identical in syntax and semantics to the corresponding
fields of a NEW_CONNECTION_ID frame (Section 19.15). A server
that chooses a zero-length connection ID MUST NOT provide a
preferred address. Similarly, a server MUST NOT include a zero-
length connection ID in this transport parameter. A client MUST
treat violation of these requirements as a connection error of
type TRANSPORT_PARAMETER_ERROR.

```
Preferred Address {
  IPv4 Address (32),
  IPv4 Port (16),
  IPv6 Address (128),
  IPv6 Port (16),
  Connection ID Length (8),
  Connection ID (..),
  Stateless Reset Token (128),
}
```

Figure 22: Preferred Address format

active_connection_id_limit (0x0e): The active connection ID limit is
   an integer value specifying the maximum number of connection IDs
   from the peer that an endpoint is willing to store. This value
   includes the connection ID received during the handshake, that
   received in the preferred_address transport parameter, and those
   received in NEW_CONNECTION_ID frames. The value of the
   active_connection_id_limit parameter MUST be at least 2. An
   endpoint that receives a value less than 2 MUST close the
   connection with an error of type TRANSPORT_PARAMETER_ERROR. If
   this transport parameter is absent, a default of 2 is assumed. If
   an endpoint issues a zero-length connection ID, it will never send
   a NEW_CONNECTION_ID frame and therefore ignores the
   active_connection_id_limit value received from its peer.

initial_source_connection_id (0x0f): The value that the endpoint
   included in the Source Connection ID field of the first Initial
   packet it sends for the connection; see Section 7.3.

retry_source_connection_id (0x10): The value that the server
   included in the Source Connection ID field of a Retry packet; see
   Section 7.3. This transport parameter is only sent by a server.

If present, transport parameters that set initial flow control limits
(initial_max_stream_data_bidi_local,
initial_max_stream_data_bidi_remote, and initial_max_stream_data_uni)
are equivalent to sending a MAX_STREAM_DATA frame (Section 19.10) on
every stream of the corresponding type immediately after opening. If
the transport parameter is absent, streams of that type start with a
flow control limit of 0.

A client MUST NOT include any server-only transport parameter:
original_destination_connection_id, preferred_address,
retry_source_connection_id, or stateless_reset_token. A server MUST
treat receipt of any of these transport parameters as a connection
error of type TRANSPORT_PARAMETER_ERROR.

19. Frame Types and Formats

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

19.1. PADDING Frames

A 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.

PADDING frames are formatted as shown in Figure 23, which shows that
PADDING frames have no content. That is, a PADDING frame consists of
the single byte that identifies the frame as a PADDING frame.

```
PADDING Frame {
  Type (i) = 0x00,
}
```

                    Figure 23: PADDING Frame Format

19.2. PING Frames

Endpoints can use PING frames (type=0x01) to verify that their peers
are still alive or to check reachability to the peer.

PING frames are formatted as shown in Figure 24, which shows that
PING frames have no content.

```
PING Frame {
  Type (i) = 0x01,
}
```

                        Figure 24: PING Frame Format

   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; see Section 10.1.2.

19.3. ACK Frames

   Receivers send ACK frames (types 0x02 and 0x03) to inform senders of
   packets they have received and processed. The ACK frame contains one
   or more ACK Ranges. ACK Ranges identify acknowledged packets. If
   the frame type is 0x03, ACK frames also contain the sum of QUIC
   packets with associated ECN marks received on the connection up until
   this point. QUIC implementations MUST properly handle both types
   and, if they have enabled ECN for packets they send, they SHOULD use
   the information in the ECN section to manage their congestion state.

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

   Packets from different packet number spaces can be identified using
   the same numeric value. An acknowledgment for a packet needs to
   indicate both a packet number and a packet number space. This is
   accomplished by having each ACK frame only acknowledge packet numbers
   in the same space as the packet in which the ACK frame is contained.

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

   ACK frames are formatted as shown in Figure 25.

   ACK Frame {
     Type (i) = 0x02..0x03,
     Largest Acknowledged (i),
     ACK Delay (i),
     ACK Range Count (i),
     First ACK Range (i),
     ACK Range (..) ...,
     [ECN Counts (..)],
   }

                        Figure 25: ACK Frame Format

ACK frames contain the following fields:

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

ACK Delay: A variable-length integer encoding the acknowledgement
   delay in microseconds; see Section 13.2.5. It is decoded by
   multiplying the value in the field by 2 to the power of the
   ack_delay_exponent transport parameter sent by the sender of the
   ACK frame; see Section 18.2. Compared to simply expressing the
   delay as an integer, this encoding allows for a larger range of
   values within the same number of bytes, at the cost of lower
   resolution.

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

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

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

ECN Counts: The three ECN Counts; see Section 19.3.2.

19.3.1. ACK Ranges

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

ACK Ranges are structured as shown in Figure 26.

```
ACK Range {
  Gap (i),
  ACK Range Length (i),
}
```

Figure 26: ACK Ranges

The fields that form each ACK Range are:

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

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

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

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

    smallest = largest - ack_range

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

The largest value for an ACK Range is determined by cumulatively
subtracting the size of all preceding ACK Ranges and Gaps.

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

The value of the Gap field establishes the largest packet number
value for the subsequent ACK Range using the following formula:

    largest = previous_smallest - gap - 2

If any computed packet number is negative, an endpoint MUST generate
a connection error of type FRAME_ENCODING_ERROR.

19.3.2. ECN Counts

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

   When present, there are 3 ECN counts, as shown in Figure 27.

   ECN Counts {
     ECT0 Count (i),
     ECT1 Count (i),
     ECN-CE Count (i),
   }

                      Figure 27: ECN Count Format

   The three ECN Counts are:

   ECT0 Count: A variable-length integer representing the total number
      of packets received with the ECT(0) codepoint in the packet number
      space of the ACK frame.

   ECT1 Count: A variable-length integer representing the total number
      of packets received with the ECT(1) codepoint in the packet number
      space of the ACK frame.

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

   ECN counts are maintained separately for each packet number space.

19.4. RESET_STREAM Frames

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

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

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

   RESET_STREAM frames are formatted as shown in Figure 28.

```
RESET_STREAM Frame {
  Type (i) = 0x04,
  Stream ID (i),
  Application Protocol Error Code (i),
  Final Size (i),
}
```

Figure 28: RESET_STREAM Frame Format

RESET_STREAM frames contain the following fields:

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

Application Protocol Error Code: A variable-length integer
   containing the application protocol error code (see Section 20.2)
   that indicates why the stream is being closed.

Final Size: A variable-length integer indicating the final size of
   the stream by the RESET_STREAM sender, in unit of bytes; see
   Section 4.5.

19.5. STOP_SENDING Frames

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

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

   STOP_SENDING frames are formatted as shown in Figure 29.

```
STOP_SENDING Frame {
  Type (i) = 0x05,
  Stream ID (i),
  Application Protocol Error Code (i),
}
```

Figure 29: STOP_SENDING Frame Format

STOP_SENDING frames contain the following fields:

Stream ID: A variable-length integer carrying the Stream ID of the

      stream being ignored.

   Application Protocol Error Code: A variable-length integer
      containing the application-specified reason the sender is ignoring
      the stream; see Section 20.2.

19.6. CRYPTO Frames

   A CRYPTO frame (type=0x06) is used to transmit cryptographic
   handshake messages. It can be sent in all packet types except 0-RTT.
   The CRYPTO frame offers the cryptographic protocol an in-order stream
   of bytes. CRYPTO frames are functionally identical to STREAM frames,
   except that they do not bear a stream identifier; they are not flow
   controlled; and they do not carry markers for optional offset,
   optional length, and the end of the stream.

   CRYPTO frames are formatted as shown in Figure 30.

   CRYPTO Frame {
     Type (i) = 0x06,
     Offset (i),
     Length (i),
     Crypto Data (..),
   }

                     Figure 30: CRYPTO Frame Format

   CRYPTO frames contain the following fields:

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

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

   Crypto Data: The cryptographic message data.

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

   The largest offset delivered on a stream - the sum of the offset and
   data length - cannot exceed $2^{62}-1$. Receipt of a frame that exceeds
   this limit MUST be treated as a connection error of type
   FRAME_ENCODING_ERROR or CRYPTO_BUFFER_EXCEEDED.

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

19.7. NEW_TOKEN Frames

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

   NEW_TOKEN frames are formatted as shown in Figure 31.

```
NEW_TOKEN Frame {
  Type (i) = 0x07,
  Token Length (i),
  Token (..),
}
```

                   Figure 31: NEW_TOKEN Frame Format

   NEW_TOKEN frames contain the following fields:

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

   Token: An opaque blob that the client may use with a future Initial
      packet. The token MUST NOT be empty. An endpoint MUST treat
      receipt of a NEW_TOKEN frame with an empty Token field as a
      connection error of type FRAME_ENCODING_ERROR.

   An endpoint might receive multiple NEW_TOKEN frames that contain the
   same token value if packets containing the frame are incorrectly
   determined to be lost. Endpoints are responsible for discarding
   duplicate values, which might be used to link connection attempts;
   see Section 8.1.3.

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

19.8. STREAM Frames

   STREAM frames implicitly create a stream and carry stream data. The
   STREAM frame Type field takes the form 0b00001XXX (or the set of
   values from 0x08 to 0x0f). 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 bytes 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) indicates that the frame marks the end of the
    stream. The final size of the stream is the sum of the offset and
    the length of this frame.

An endpoint MUST terminate the connection with error
STREAM_STATE_ERROR if it receives a STREAM frame for a locally-
initiated stream that has not yet been created, or for a send-only
stream.

STREAM frames are formatted as shown in Figure 32.

```
STREAM Frame {
  Type (i) = 0x08..0x0f,
  Stream ID (i),
  [Offset (i)],
  [Length (i)],
  Stream Data (..),
}
```

Figure 32: STREAM Frame Format

STREAM frames contain the following fields:

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

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

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

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

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

   The first byte in the stream has an offset of 0. The largest offset
   delivered on a stream - the sum of the offset and data length -
   cannot exceed 2^62-1, as it is not possible to provide flow control
   credit for that data. Receipt of a frame that exceeds this limit
   MUST be treated as a connection error of type FRAME_ENCODING_ERROR or
   FLOW_CONTROL_ERROR.

19.9. MAX_DATA Frames

   A MAX_DATA frame (type=0x10) is used in flow control to inform the
   peer of the maximum amount of data that can be sent on the connection
   as a whole.

   MAX_DATA frames are formatted as shown in Figure 33.

```
MAX_DATA Frame {
  Type (i) = 0x10,
  Maximum Data (i),
}
```

                      Figure 33: MAX_DATA Frame Format

   MAX_DATA frames contain the following field:

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

   All data sent in STREAM frames counts toward this limit. The sum of
   the final sizes on all streams - including streams in terminal states
   - MUST NOT exceed the value advertised by a receiver. An endpoint
   MUST terminate a connection with a FLOW_CONTROL_ERROR error if it
   receives more data than the maximum data value that it has sent.
   This includes violations of remembered limits in Early Data; see
   Section 7.4.1.

19.10. MAX_STREAM_DATA Frames

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

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

MAX_STREAM_DATA frames are formatted as shown in Figure 34.

```
MAX_STREAM_DATA Frame {
  Type (i) = 0x11,
  Stream ID (i),
  Maximum Stream Data (i),
}
```

                  Figure 34: MAX_STREAM_DATA Frame Format

MAX_STREAM_DATA frames contain the following fields:

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

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

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

The data sent on a stream MUST NOT exceed the largest maximum stream
data value advertised by the receiver. An endpoint MUST terminate a
connection with a FLOW_CONTROL_ERROR error if it receives more data
than the largest maximum stream data that it has sent for the
affected stream. This includes violations of remembered limits in
Early Data; see Section 7.4.1.

19.11. MAX_STREAMS Frames

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

MAX_STREAMS frames are formatted as shown in Figure 35;

```
MAX_STREAMS Frame {
  Type (i) = 0x12..0x13,
  Maximum Streams (i),
}
```

Figure 35: MAX_STREAMS Frame Format

MAX_STREAMS frames contain the following field:

Maximum Streams: A count of the cumulative number of streams of the
   corresponding type that can be opened over the lifetime of the
   connection. This value cannot exceed 2^60, as it is not possible
   to encode stream IDs larger than 2^62-1. Receipt of a frame that
   permits opening of a stream larger than this limit MUST be treated
   as a FRAME_ENCODING_ERROR.

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

An endpoint MUST NOT open more streams than permitted by the current
stream limit set by its peer. For instance, a server that receives a
unidirectional stream limit of 3 is permitted to open stream 3, 7,
and 11, but not stream 15. An endpoint MUST terminate a connection
with a STREAM_LIMIT_ERROR error if a peer opens more streams than was
permitted. This includes violations of remembered limits in Early
Data; see Section 7.4.1.

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

19.12. DATA_BLOCKED Frames

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

DATA_BLOCKED frames are formatted as shown in Figure 36.

```
DATA_BLOCKED Frame {
  Type (i) = 0x14,
  Maximum Data (i),
}
```

Figure 36: DATA_BLOCKED Frame Format

DATA_BLOCKED frames contain the following field:

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

19.13. STREAM_DATA_BLOCKED Frames

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

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

STREAM_DATA_BLOCKED frames are formatted as shown in Figure 37.

```
STREAM_DATA_BLOCKED Frame {
  Type (i) = 0x15,
  Stream ID (i),
  Maximum Stream Data (i),
}
```

Figure 37: STREAM_DATA_BLOCKED Frame Format

STREAM_DATA_BLOCKED frames contain the following fields:

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

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

19.14. STREAMS_BLOCKED Frames

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

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

STREAMS_BLOCKED frames are formatted as shown in Figure 38.

```
STREAMS_BLOCKED Frame {
  Type (i) = 0x16..0x17,
  Maximum Streams (i),
}
```

Figure 38: STREAMS_BLOCKED Frame Format

STREAMS_BLOCKED frames contain the following field:

Maximum Streams: A variable-length integer indicating the maximum
   number of streams allowed at the time the frame was sent. This
   value cannot exceed 2^{60}, as it is not possible to encode stream
   IDs larger than $2^{62}-1$. Receipt of a frame that encodes a larger
   stream ID MUST be treated as a STREAM_LIMIT_ERROR or a
   FRAME_ENCODING_ERROR.

19.15. NEW_CONNECTION_ID Frames

An endpoint sends a NEW_CONNECTION_ID frame (type=0x18) to provide
its peer with alternative connection IDs that can be used to break
linkability when migrating connections; see Section 9.5.

NEW_CONNECTION_ID frames are formatted as shown in Figure 39.

```
NEW_CONNECTION_ID Frame {
  Type (i) = 0x18,
  Sequence Number (i),
  Retire Prior To (i),
  Length (8),
  Connection ID (8..160),
  Stateless Reset Token (128),
}
```

Figure 39: NEW_CONNECTION_ID Frame Format

NEW_CONNECTION_ID frames contain the following fields:

Sequence Number: The sequence number assigned to the connection ID
   by the sender, encoded as a variable-length integer; see
   Section 5.1.1.

Retire Prior To: A variable-length integer indicating which

   connection IDs should be retired; see Section 5.1.2.

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

   Connection ID: A connection ID of the specified length.

   Stateless Reset Token: A 128-bit value that will be used for a
      stateless reset when the associated connection ID is used; see
      Section 10.3.

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

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

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

   The Retire Prior To field applies to connection IDs established
   during connection setup and the preferred_address transport
   parameter; see Section 5.1.2. The Retire Prior To field MUST be less
   than or equal to the Sequence Number field. Receiving a value
   greater than the Sequence Number MUST be treated as a connection
   error of type FRAME_ENCODING_ERROR.

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

An endpoint that receives a NEW_CONNECTION_ID frame with a sequence
number smaller than the Retire Prior To field of a previously
received NEW_CONNECTION_ID frame MUST send a corresponding
RETIRE_CONNECTION_ID frame that retires the newly received connection
ID, unless it has already done so for that sequence number.

19.16. RETIRE_CONNECTION_ID Frames

An endpoint sends a RETIRE_CONNECTION_ID frame (type=0x19) to
indicate that it will no longer use a connection ID that was issued
by its peer. This may include the connection ID provided during the
handshake. Sending a RETIRE_CONNECTION_ID frame also serves as a
request to the peer to send additional connection IDs for future use;
see Section 5.1. New connection IDs can be delivered to a peer using
the NEW_CONNECTION_ID frame (Section 19.15).

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

RETIRE_CONNECTION_ID frames are formatted as shown in Figure 40.

```
RETIRE_CONNECTION_ID Frame {
  Type (i) = 0x19,
  Sequence Number (i),
}
```

Figure 40: RETIRE_CONNECTION_ID Frame Format

RETIRE_CONNECTION_ID frames contain the following field:

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

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

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

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

19.17. PATH_CHALLENGE Frames

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

   PATH_CHALLENGE frames are formatted as shown in Figure 41.

   PATH_CHALLENGE Frame {
     Type (i) = 0x1a,
     Data (64),
   }

                   Figure 41: PATH_CHALLENGE Frame Format

   PATH_CHALLENGE frames contain the following field:

   Data: This 8-byte field contains arbitrary data.

   Including 64 bits of entropy in a PATH_CHALLENGE frame ensures that
   it is easier to receive the packet than it is to guess the value
   correctly.

   The recipient of this frame MUST generate a PATH_RESPONSE frame
   (Section 19.18) containing the same Data.

19.18. PATH_RESPONSE Frames

   A PATH_RESPONSE frame (type=0x1b) is sent in response to a
   PATH_CHALLENGE frame.

   PATH_RESPONSE frames are formatted as shown in Figure 42, which is
   identical to the PATH_CHALLENGE frame (Section 19.17).

   PATH_RESPONSE Frame {
     Type (i) = 0x1b,
     Data (64),
   }

                   Figure 42: PATH_RESPONSE Frame Format

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

19.19. CONNECTION_CLOSE Frames

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

   If there are open streams that have not been explicitly closed, they
   are implicitly closed when the connection is closed.

   CONNECTION_CLOSE frames are formatted as shown in Figure 43.

```
CONNECTION_CLOSE Frame {
  Type (i) = 0x1c..0x1d,
  Error Code (i),
  [Frame Type (i)],
  Reason Phrase Length (i),
  Reason Phrase (..),
}
```

                  Figure 43: CONNECTION_CLOSE Frame Format

   CONNECTION_CLOSE frames contain the following fields:

   Error Code: A variable-length integer error code that indicates the
      reason for closing this connection. A CONNECTION_CLOSE frame of
      type 0x1c uses codes from the space defined in Section 20.1. A
      CONNECTION_CLOSE frame of type 0x1d uses codes from the
      application protocol error code space; see Section 20.2.

   Frame Type: A variable-length integer encoding the type of frame
      that triggered the error. A value of 0 (equivalent to the mention
      of the PADDING frame) is used when the frame type is unknown. The
      application-specific variant of CONNECTION_CLOSE (type 0x1d) does
      not include this field.

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

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

The application-specific variant of CONNECTION_CLOSE (type 0x1d) can
only be sent using 0-RTT or 1-RTT packets; see Section 12.5. When an
application wishes to abandon a connection during the handshake, an
endpoint can send a CONNECTION_CLOSE frame (type 0x1c) with an error
code of APPLICATION_ERROR in an Initial or a Handshake packet.

19.20. HANDSHAKE_DONE Frames

The server uses a HANDSHAKE_DONE frame (type=0x1e) to signal
confirmation of the handshake to the client.

HANDSHAKE_DONE frames are formatted as shown in Figure 44, which
shows that HANDSHAKE_DONE frames have no content.

```
HANDSHAKE_DONE Frame {
  Type (i) = 0x1e,
}
```

Figure 44: HANDSHAKE_DONE Frame Format

A HANDSHAKE_DONE frame can only be sent by the server. Servers MUST
NOT send a HANDSHAKE_DONE frame before completing the handshake. A
server MUST treat receipt of a HANDSHAKE_DONE frame as a connection
error of type PROTOCOL_VIOLATION.

19.21. Extension Frames

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

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

Extensions that modify or replace core protocol functionality
(including frame types) will be difficult to combine with other
extensions that modify or replace the same functionality unless the
behavior of the combination is explicitly defined. Such extensions
SHOULD define their interaction with previously-defined extensions
modifying the same protocol components.

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

An IANA registry is used to manage the assignment of frame types; see
Section 22.3.

20. Error Codes

QUIC transport error codes and application error codes are 62-bit
unsigned integers.

20.1. Transport Error Codes

This section lists the defined QUIC transport error codes that may be
used in a CONNECTION_CLOSE frame with a type of 0x1c. 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.

CONNECTION_REFUSED (0x2): The server refused to accept a new
   connection.

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

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

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

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

FRAME_ENCODING_ERROR (0x7): An endpoint received a frame that was

   badly formatted. For instance, a frame of an unknown type, or an
   ACK frame that has more acknowledgment ranges than the remainder
   of the packet could carry.

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

CONNECTION_ID_LIMIT_ERROR (0x9): The number of connection IDs
   provided by the peer exceeds the advertised
   active_connection_id_limit.

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

INVALID_TOKEN (0xb): A server received a client Initial that
   contained an invalid Token field.

APPLICATION_ERROR (0xc): The application or application protocol
   caused the connection to be closed.

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

KEY_UPDATE_ERROR (0xe): An endpoint detected errors in performing
   key updates; see Section 6 of [QUIC-TLS].

AEAD_LIMIT_REACHED (0xf): An endpoint has reached the
   confidentiality or integrity limit for the AEAD algorithm used by
   the given connection.

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

   See Section 22.4 for details of registering new error codes.

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

20.2. Application Protocol Error Codes

The management of application error codes is left to application
protocols. Application protocol error codes are used for the
RESET_STREAM frame (Section 19.4), the STOP_SENDING frame
(Section 19.5), and the CONNECTION_CLOSE frame with a type of 0x1d
(Section 19.19).

21. Security Considerations

21.1. Handshake Denial of Service

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

Once a connection is established QUIC endpoints might accept some
unauthenticated ICMP packets (see Section 14.2.1), but the use of
these packets is extremely limited. The only other type of packet
that an endpoint might accept is a stateless reset (Section 10.3),
which relies on the token being kept secret until it is used.

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

Addresses cannot change during the handshake, so endpoints can
discard packets that are received on a different network path.

The Source and Destination Connection ID fields are the primary means
of protection against off-path attack during the handshake. These
are required to match those set by a peer. Except for an Initial and
stateless reset packets, an endpoint only accepts packets that
include a Destination Connection ID field that matches a value the
endpoint previously chose. This is the only protection offered for
Version Negotiation packets.

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

These protections are not intended to be effective against an attacker that is able to receive QUIC packets prior to the connection being established. Such an attacker can potentially send packets that will be accepted by QUIC endpoints. This version of QUIC attempts to detect this sort of attack, but it expects that endpoints will fail to establish a connection rather than recovering. For the most part, the cryptographic handshake protocol [QUIC-TLS] is responsible for detecting tampering during the handshake.

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

21.2. Amplification Attack

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

Servers SHOULD provide mitigations for this attack by limiting the usage and lifetime of address validation tokens; see Section 8.1.3.

21.3. Optimistic ACK Attack

An endpoint that acknowledges packets it has not received might cause a congestion controller to permit sending at rates beyond what the network supports. An endpoint MAY skip packet numbers when sending packets to detect this behavior. An endpoint can then immediately close the connection with a connection error of type PROTOCOL_VIOLATION; see Section 10.2.

21.4. Request Forgery Attacks

   A request forgery attack occurs where an endpoint causes its peer to
   issue a request towards a victim, with the request controlled by the
   endpoint. Request forgery attacks aim to provide an attacker with
   access to capabilities of its peer that might otherwise be
   unavailable to the attacker. For a networking protocol, a request
   forgery attack is often used to exploit any implicit authorization
   conferred on the peer by the victim due to the peer's location in the
   network.

   For request forgery to be effective, an attacker needs to be able to
   influence what packets the peer sends and where these packets are
   sent. If an attacker can target a vulnerable service with a
   controlled payload, that service might perform actions that are
   attributed to the attacker's peer, but decided by the attacker.

   For example, cross-site request forgery [CSRF] exploits on the Web
   cause a client to issue requests that include authorization cookies
   [COOKIE], allowing one site access to information and actions that
   are intended to be restricted to a different site.

   As QUIC runs over UDP, the primary attack modality of concern is one
   where an attacker can select the address to which its peer sends UDP
   datagrams and can control some of the unprotected content of those
   packets. As much of the data sent by QUIC endpoints is protected,
   this includes control over ciphertext. An attack is successful if an
   attacker can cause a peer to send a UDP datagram to a host that will
   perform some action based on content in the datagram.

   This section discusses ways in which QUIC might be used for request
   forgery attacks.

   This section also describes limited countermeasures that can be
   implemented by QUIC endpoints. These mitigations can be employed
   unilaterally by a QUIC implementation or deployment, without
   potential targets for request forgery attacks taking action. However
   these countermeasures could be insufficient if UDP-based services do
   not properly authorize requests.

   Because the migration attack described in Section 21.4.4 is quite
   powerful and does not have adequate countermeasures, QUIC server
   implementations should assume that attackers can cause them to
   generate arbitrary UDP payloads to arbitrary destinations. QUIC
   servers SHOULD NOT be deployed in networks that also have
   inadequately secured UDP endpoints.

Although it is not generally possible to ensure that clients are not
co-located with vulnerable endpoints, this version of QUIC does not
allow servers to migrate, thus preventing spoofed migration attacks
on clients. Any future extension which allows server migration MUST
also define countermeasures for forgery attacks.

21.4.1. Control Options for Endpoints

QUIC offers some opportunities for an attacker to influence or
control where its peer sends UDP datagrams:

* initial connection establishment (Section 7), where a server is
   able to choose where a client sends datagrams, for example by
   populating DNS records;

* preferred addresses (Section 9.6), where a server is able to
   choose where a client sends datagrams; and

* spoofed connection migrations (Section 9.3.1), where a client is
   able to use source address spoofing to select where a server sends
   subsequent datagrams.

In all three cases, the attacker can cause its peer to send datagrams
to a victim that might not understand QUIC. That is, these packets
are sent by the peer prior to address validation; see Section 8.

Outside of the encrypted portion of packets, QUIC offers an endpoint
several options for controlling the content of UDP datagrams that its
peer sends. The Destination Connection ID field offers direct
control over bytes that appear early in packets sent by the peer; see
Section 5.1. The Token field in Initial packets offers a server
control over other bytes of Initial packets; see Section 17.2.2.

There are no measures in this version of QUIC to prevent indirect
control over the encrypted portions of packets. It is necessary to
assume that endpoints are able to control the contents of frames that
a peer sends, especially those frames that convey application data,
such as STREAM frames. Though this depends to some degree on details
of the application protocol, some control is possible in many
protocol usage contexts. As the attacker has access to packet
protection keys, they are likely to be capable of predicting how a
peer will encrypt future packets. Successful control over datagram
content then only requires that the attacker be able to predict the
packet number and placement of frames in packets with some amount of
reliability.

This section assumes that limiting control over datagram content is
not feasible. The focus of the mitigations in subsequent sections is
on limiting the ways in which datagrams that are sent prior to
address validation can be used for request forgery.

21.4.2. Request Forgery with Client Initial Packets

An attacker acting as a server can choose the IP address and port on
which it advertises its availability, so Initial packets from clients
are assumed to be available for use in this sort of attack. The
address validation implicit in the handshake ensures that - for a new
connection - a client will not send other types of packet to a
destination that does not understand QUIC or is not willing to accept
a QUIC connection.

Initial packet protection (Section 5.2 of [QUIC-TLS]) makes it
difficult for servers to control the content of Initial packets sent
by clients. A client choosing an unpredictable Destination
Connection ID ensures that servers are unable to control any of the
encrypted portion of Initial packets from clients.

However, the Token field is open to server control and does allow a
server to use clients to mount request forgery attacks. Use of
tokens provided with the NEW_TOKEN frame (Section 8.1.3) offers the
only option for request forgery during connection establishment.

Clients however are not obligated to use the NEW_TOKEN frame.
Request forgery attacks that rely on the Token field can be avoided
if clients send an empty Token field when the server address has
changed from when the NEW_TOKEN frame was received.

Clients could avoid using NEW_TOKEN if the server address changes.
However, not including a Token field could adversely affect
performance. Servers could rely on NEW_TOKEN to enable sending of
data in excess of the three times limit on sending data; see
Section 8.1. In particular, this affects cases where clients use
0-RTT to request data from servers.

Sending a Retry packet (Section 17.2.5) offers a server the option to
change the Token field. After sending a Retry, the server can also
control the Destination Connection ID field of subsequent Initial
packets from the client. This also might allow indirect control over
the encrypted content of Initial packets. However, the exchange of a
Retry packet validates the server's address, thereby preventing the
use of subsequent Initial packets for request forgery.

21.4.3. Request Forgery with Preferred Addresses

   Servers can specify a preferred address, which clients then migrate
   to after confirming the handshake; see Section 9.6. The Destination
   Connection ID field of packets that the client sends to a preferred
   address can be used for request forgery.

   A client MUST NOT send non-probing frames to a preferred address
   prior to validating that address; see Section 8. This greatly
   reduces the options that a server has to control the encrypted
   portion of datagrams.

   This document does not offer any additional countermeasures that are
   specific to use of preferred addresses and can be implemented by
   endpoints. The generic measures described in Section 21.4.5 could be
   used as further mitigation.

21.4.4. Request Forgery with Spoofed Migration

   Clients are able to present a spoofed source address as part of an
   apparent connection migration to cause a server to send datagrams to
   that address.

   The Destination Connection ID field in any packets that a server
   subsequently sends to this spoofed address can be used for request
   forgery. A client might also be able to influence the ciphertext.

   A server that only sends probing packets (Section 9.1) to an address
   prior to address validation provides an attacker with only limited
   control over the encrypted portion of datagrams. However,
   particularly for NAT rebinding, this can adversely affect
   performance. If the server sends frames carrying application data,
   an attacker might be able to control most of the content of
   datagrams.

   This document does not offer specific countermeasures that can be
   implemented by endpoints aside from the generic measures described in
   Section 21.4.5. However, countermeasures for address spoofing at the
   network level, in particular ingress filtering [BCP38], are
   especially effective against attacks that use spoofing and originate
   from an external network.

21.4.5. Generic Request Forgery Countermeasures

   The most effective defense against request forgery attacks is to
   modify vulnerable services to use strong authentication. However,
   this is not always something that is within the control of a QUIC
   deployment. This section outlines some others steps that QUIC
   endpoints could take unilaterally. These additional steps are all
   discretionary as, depending on circumstances, they could interfere
   with or prevent legitimate uses.

   Services offered over loopback interfaces often lack proper
   authentication. Endpoints MAY prevent connection attempts or
   migration to a loopback address. Endpoints SHOULD NOT allow
   connections or migration to a loopback address if the same service
   was previously available at a different interface or if the address
   was provided by a service at a non-loopback address. Endpoints that
   depend on these capabilities could offer an option to disable these
   protections.

   Similarly, endpoints could regard a change in address to link-local
   address [RFC4291] or an address in a private use range [RFC1918] from
   a global, unique-local [RFC4193], or non-private address as a
   potential attempt at request forgery. Endpoints could refuse to use
   these addresses entirely, but that carries a significant risk of
   interfering with legitimate uses. Endpoints SHOULD NOT refuse to use
   an address unless they have specific knowledge about the network
   indicating that sending datagrams to unvalidated addresses in a given
   range is not safe.

   Endpoints MAY choose to reduce the risk of request forgery by not
   including values from NEW_TOKEN frames in Initial packets or by only
   sending probing frames in packets prior to completing address
   validation. Note that this does not prevent an attacker from using
   the Destination Connection ID field for an attack.

   Endpoints are not expected to have specific information about the
   location of servers that could be vulnerable targets of a request
   forgery attack. However, it might be possible over time to identify
   specific UDP ports that are common targets of attacks or particular
   patterns in datagrams that are used for attacks. Endpoints MAY
   choose to avoid sending datagrams to these ports or not send
   datagrams that match these patterns prior to validating the
   destination address. Endpoints MAY retire connection IDs containing
   patterns known to be problematic without using them.

   Note: Modifying endpoints to apply these protections is more

efficient than deploying network-based protections, as endpoints
do not need to perform any additional processing when sending to
an address that has been validated.

21.5. Slowloris Attacks

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

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

21.6. Stream Fragmentation and Reassembly Attacks

An adversarial sender might intentionally not send portions of the
stream data, causing the receiver to commit resources for the unsent
data. This could cause a disproportionate receive buffer memory
commitment and/or the creation of a large and inefficient data
structure at the receiver.

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

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

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

21.7. Stream Commitment Attack

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

   Normally, clients will open streams sequentially, as explained in
   Section 2.1. However, when several streams are initiated at short
   intervals, loss or reordering may cause STREAM frames that open
   streams to be received out of sequence. On receiving a higher-
   numbered stream ID, a receiver is required to open all intervening
   streams of the same type; see Section 3.2. Thus, on a new
   connection, opening stream 4000000 opens 1 million and 1 client-
   initiated bidirectional streams.

   The number of active streams is limited by the
   initial_max_streams_bidi and initial_max_streams_uni transport
   parameters, as explained in Section 4.6. If chosen judiciously,
   these limits mitigate the effect of the stream commitment attack.
   However, setting the limit too low could affect performance when
   applications expect to open large number of streams.

21.8. Peer Denial of Service

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

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

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

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

21.9. Explicit Congestion Notification Attacks

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

   An on-the-side attacker can duplicate and send packets with modified
   ECN fields to affect the sender's rate. If duplicate packets are
   discarded by a receiver, an off-path attacker will need to race the
   duplicate packet against the original to be successful in this
   attack. Therefore, QUIC endpoints ignore the ECN field on an IP
   packet unless at least one QUIC packet in that IP packet is
   successfully processed; see Section 13.4.

21.10. Stateless Reset Oracle

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

   If a packet can be routed to different instances that share a static
   key, for example by changing an IP address or port, then an attacker
   can cause the server to send a stateless reset. To defend against
   this style of denial of service, endpoints that share a static key
   for stateless reset (see Section 10.3.2) MUST be arranged so that
   packets with a given connection ID always arrive at an instance that
   has connection state, unless that connection is no longer active.

   More generally, servers MUST NOT generate a stateless reset if a
   connection with the corresponding connection ID could be active on
   any endpoint using the same static key.

   In the case of a cluster that uses dynamic load balancing, it is
   possible that a change in load balancer configuration could occur
   while an active instance retains connection state. Even if an
   instance retains connection state, the change in routing and
   resulting stateless reset will result in the connection being
   terminated. If there is no chance of the packet being routed to the
   correct instance, it is better to send a stateless reset than wait
   for the connection to time out. However, this is acceptable only if
   the routing cannot be influenced by an attacker.

21.11. Version Downgrade

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

21.12. Targeted Attacks by Routing

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

21.13. Overview of Security Properties

   A complete security analysis of QUIC is outside the scope of this
   document. This section provides an informal description of the
   desired security properties as an aid to implementors and to help
   guide protocol analysis.

   QUIC assumes the threat model described in [SEC-CONS] and provides
   protections against many of the attacks that arise from that model.

   For this purpose, attacks are divided into passive and active
   attacks. Passive attackers have the capability to read packets from
   the network, while active attackers also have the capability to write
   packets into the network. However, a passive attack may involve an
   attacker with the ability to cause a routing change or other
   modification in the path taken by packets that comprise a connection.

   Attackers are additionally categorized as either on-path attackers or
   off-path attackers; see Section 3.5 of [SEC-CONS]. An on-path
   attacker can read, modify, or remove any packet it observes such that
   it no longer reaches its destination, while an off-path attacker
   observes the packets, but cannot prevent the original packet from
   reaching its intended destination. Both types of attackers can also
   transmit arbitrary packets.

   Properties of the handshake, protected packets, and connection
   migration are considered separately.

21.13.1. Handshake

The QUIC handshake incorporates the TLS 1.3 handshake and inherits
the cryptographic properties described in Appendix E.1 of [TLS13].
Many of the security properties of QUIC depend on the TLS handshake
providing these properties. Any attack on the TLS handshake could
affect QUIC.

Any attack on the TLS handshake that compromises the secrecy or
uniqueness of session keys affects other security guarantees provided
by QUIC that depends on these keys. For instance, migration
(Section 9) depends on the efficacy of confidentiality protections,
both for the negotiation of keys using the TLS handshake and for QUIC
packet protection, to avoid linkability across network paths.

An attack on the integrity of the TLS handshake might allow an
attacker to affect the selection of application protocol or QUIC
version.

In addition to the properties provided by TLS, the QUIC handshake
provides some defense against DoS attacks on the handshake.

21.13.1.1. Anti-Amplification

Address validation (Section 8) is used to verify that an entity that
claims a given address is able to receive packets at that address.
Address validation limits amplification attack targets to addresses
for which an attacker can observe packets.

Prior to validation, endpoints are limited in what they are able to
send. During the handshake, a server cannot send more than three
times the data it receives; clients that initiate new connections or
migrate to a new network path are limited.

21.13.1.2. Server-Side DoS

Computing the server's first flight for a full handshake is
potentially expensive, requiring both a signature and a key exchange
computation. In order to prevent computational DoS attacks, the
Retry packet provides a cheap token exchange mechanism that allows
servers to validate a client's IP address prior to doing any
expensive computations at the cost of a single round trip. After a
successful handshake, servers can issue new tokens to a client, which
will allow new connection establishment without incurring this cost.

21.13.1.3. On-Path Handshake Termination

An on-path or off-path attacker can force a handshake to fail by
replacing or racing Initial packets. Once valid Initial packets have
been exchanged, subsequent Handshake packets are protected with the
handshake keys and an on-path attacker cannot force handshake failure
other than by dropping packets to cause endpoints to abandon the
attempt.

An on-path attacker can also replace the addresses of packets on
either side and therefore cause the client or server to have an
incorrect view of the remote addresses. Such an attack is
indistinguishable from the functions performed by a NAT.

21.13.1.4. Parameter Negotiation

The entire handshake is cryptographically protected, with the Initial
packets being encrypted with per-version keys and the Handshake and
later packets being encrypted with keys derived from the TLS key
exchange. Further, parameter negotiation is folded into the TLS
transcript and thus provides the same integrity guarantees as
ordinary TLS negotiation. An attacker can observe the client's
transport parameters (as long as it knows the version-specific salt)
but cannot observe the server's transport parameters and cannot
influence parameter negotiation.

Connection IDs are unencrypted but integrity protected in all
packets.

This version of QUIC does not incorporate a version negotiation
mechanism; implementations of incompatible versions will simply fail
to establish a connection.

21.13.2. Protected Packets

Packet protection (Section 12.1) provides authentication and
encryption of all packets except Version Negotiation packets, though
Initial and Retry packets have limited encryption and authentication
based on version-specific inputs; see [QUIC-TLS] for more details.
This section considers passive and active attacks against protected
packets.

Both on-path and off-path attackers can mount a passive attack in
which they save observed packets for an offline attack against packet
protection at a future time; this is true for any observer of any
packet on any network.

   A blind attacker, one who injects packets without being able to
   observe valid packets for a connection, is unlikely to be successful,
   since packet protection ensures that valid packets are only generated
   by endpoints that possess the key material established during the
   handshake; see Section 7 and Section 21.13.1. Similarly, any active
   attacker that observes packets and attempts to insert new data or
   modify existing data in those packets should not be able to generate
   packets deemed valid by the receiving endpoint.

   A spoofing attack, in which an active attacker rewrites unprotected
   parts of a packet that it forwards or injects, such as the source or
   destination address, is only effective if the attacker can forward
   packets to the original endpoint. Packet protection ensures that the
   packet payloads can only be processed by the endpoints that completed
   the handshake, and invalid packets are ignored by those endpoints.

   An attacker can also modify the boundaries between packets and UDP
   datagrams, causing multiple packets to be coalesced into a single
   datagram, or splitting coalesced packets into multiple datagrams.
   Aside from datagrams containing Initial packets, which require
   padding, modification of how packets are arranged in datagrams has no
   functional effect on a connection, although it might change some
   performance characteristics.

21.13.3. Connection Migration

   Connection Migration (Section 9) provides endpoints with the ability
   to transition between IP addresses and ports on multiple paths, using
   one path at a time for transmission and receipt of non-probing
   frames. Path validation (Section 8.2) establishes that a peer is
   both willing and able to receive packets sent on a particular path.
   This helps reduce the effects of address spoofing by limiting the
   number of packets sent to a spoofed address.

   This section describes the intended security properties of connection
   migration when under various types of DoS attacks.

21.13.3.1. On-Path Active Attacks

   An attacker that can cause a packet it observes to no longer reach
   its intended destination is considered an on-path attacker. When an
   attacker is present between a client and server, endpoints are
   required to send packets through the attacker to establish
   connectivity on a given path.

   An on-path attacker can:

   * Inspect packets

* Modify IP and UDP packet headers

* Inject new packets

* Delay packets

* Reorder packets

* Drop packets

* Split and merge datagrams along packet boundaries

An on-path attacker cannot:

* Modify an authenticated portion of a packet and cause the
   recipient to accept that packet

An on-path attacker has the opportunity to modify the packets that it
observes, however any modifications to an authenticated portion of a
packet will cause it to be dropped by the receiving endpoint as
invalid, as packet payloads are both authenticated and encrypted.

In the presence of an on-path attacker, QUIC aims to provide the
following properties:

1. An on-path attacker can prevent use of a path for a connection,
    causing it to fail if it cannot use a different path that does
    not contain the attacker. This can be achieved by dropping all
    packets, modifying them so that they fail to decrypt, or other
    methods.

2. An on-path attacker can prevent migration to a new path for which
    the attacker is also on-path by causing path validation to fail
    on the new path.

3. An on-path attacker cannot prevent a client from migrating to a
    path for which the attacker is not on-path.

4. An on-path attacker can reduce the throughput of a connection by
    delaying packets or dropping them.

5. An on-path attacker cannot cause an endpoint to accept a packet
    for which it has modified an authenticated portion of that
    packet.

21.13.3.2. Off-Path Active Attacks

   An off-path attacker is not directly on the path between a client and
   server, but could be able to obtain copies of some or all packets
   sent between the client and the server. It is also able to send
   copies of those packets to either endpoint.

   An off-path attacker can:

   * Inspect packets

   * Inject new packets

   * Reorder injected packets

   An off-path attacker cannot:

   * Modify any part of a packet

   * Delay packets

   * Drop packets

   * Reorder original packets

   An off-path attacker can modify packets that it has observed and
   inject them back into the network, potentially with spoofed source
   and destination addresses.

   For the purposes of this discussion, it is assumed that an off-path
   attacker has the ability to observe, modify, and re-inject a packet
   into the network that will reach the destination endpoint prior to
   the arrival of the original packet observed by the attacker. In
   other words, an attacker has the ability to consistently "win" a race
   with the legitimate packets between the endpoints, potentially
   causing the original packet to be ignored by the recipient.

   It is also assumed that an attacker has the resources necessary to
   affect NAT state, potentially both causing an endpoint to lose its
   NAT binding, and an attacker to obtain the same port for use with its
   traffic.

   In the presence of an off-path attacker, QUIC aims to provide the
   following properties:

   1. An off-path attacker can race packets and attempt to become a
       "limited" on-path attacker.

2. An off-path attacker can cause path validation to succeed for
    forwarded packets with the source address listed as the off-path
    attacker as long as it can provide improved connectivity between
    the client and the server.

3. An off-path attacker cannot cause a connection to close once the
    handshake has completed.

4. An off-path attacker cannot cause migration to a new path to fail
    if it cannot observe the new path.

5. An off-path attacker can become a limited on-path attacker during
    migration to a new path for which it is also an off-path
    attacker.

6. An off-path attacker can become a limited on-path attacker by
    affecting shared NAT state such that it sends packets to the
    server from the same IP address and port that the client
    originally used.

21.13.3.3. Limited On-Path Active Attacks

A limited on-path attacker is an off-path attacker that has offered
improved routing of packets by duplicating and forwarding original
packets between the server and the client, causing those packets to
arrive before the original copies such that the original packets are
dropped by the destination endpoint.

A limited on-path attacker differs from an on-path attacker in that
it is not on the original path between endpoints, and therefore the
original packets sent by an endpoint are still reaching their
destination. This means that a future failure to route copied
packets to the destination faster than their original path will not
prevent the original packets from reaching the destination.

A limited on-path attacker can:

*   Inspect packets

*   Inject new packets

*   Modify unencrypted packet headers

*   Reorder packets

A limited on-path attacker cannot:

   * Delay packets so that they arrive later than packets sent on the
      original path

   * Drop packets

   * Modify the authenticated and encrypted portion of a packet and
      cause the recipient to accept that packet

   A limited on-path attacker can only delay packets up to the point
   that the original packets arrive before the duplicate packets,
   meaning that it cannot offer routing with worse latency than the
   original path. If a limited on-path attacker drops packets, the
   original copy will still arrive at the destination endpoint.

   In the presence of a limited on-path attacker, QUIC aims to provide
   the following properties:

   1. A limited on-path attacker cannot cause a connection to close
       once the handshake has completed.

   2. A limited on-path attacker cannot cause an idle connection to
       close if the client is first to resume activity.

   3. A limited on-path attacker can cause an idle connection to be
       deemed lost if the server is the first to resume activity.

   Note that these guarantees are the same guarantees provided for any
   NAT, for the same reasons.

22. IANA Considerations

   This document establishes several registries for the management of
   codepoints in QUIC. These registries operate on a common set of
   policies as defined in Section 22.1.

22.1. Registration Policies for QUIC Registries

   All QUIC registries allow for both provisional and permanent
   registration of codepoints. This section documents policies that are
   common to these registries.

22.1.1. Provisional Registrations

   Provisional registration of codepoints are intended to allow for
   private use and experimentation with extensions to QUIC. Provisional
   registrations only require the inclusion of the codepoint value and
   contact information. However, provisional registrations could be
   reclaimed and reassigned for another purpose.

   Provisional registrations require Expert Review, as defined in
   Section 4.5 of [RFC8126]. Designated expert(s) are advised that only
   registrations for an excessive proportion of remaining codepoint
   space or the very first unassigned value (see Section 22.1.2) can be
   rejected.

   Provisional registrations will include a date field that indicates
   when the registration was last updated. A request to update the date
   on any provisional registration can be made without review from the
   designated expert(s).

   All QUIC registries include the following fields to support
   provisional registration:

   Value: The assigned codepoint.

   Status: "Permanent" or "Provisional".

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

   Date: The date of last update to the registration.

   Change Controller: The entity that is responsible for the definition
      of the registration.

   Contact: Contact details for the registrant.

   Notes: Supplementary notes about the registration.

   Provisional registrations MAY omit the Specification and Notes
   fields, plus any additional fields that might be required for a
   permanent registration. The Date field is not required as part of
   requesting a registration as it is set to the date the registration
   is created or updated.

22.1.2. Selecting Codepoints

   New uses of codepoints from QUIC registries SHOULD use a randomly
   selected codepoint that excludes both existing allocations and the
   first unallocated codepoint in the selected space. Requests for
   multiple codepoints MAY use a contiguous range. This minimizes the
   risk that differing semantics are attributed to the same codepoint by
   different implementations. Use of the first codepoint in a range is
   intended for use by specifications that are developed through the
   standards process [STD] and its allocation MUST be negotiated with
   IANA before use.

For codepoints that are encoded in variable-length integers
(Section 16), such as frame types, codepoints that encode to four or
eight bytes (that is, values 2^14 and above) SHOULD be used unless
the usage is especially sensitive to having a longer encoding.

Applications to register codepoints in QUIC registries MAY include a
codepoint as part of the registration. IANA MUST allocate the
selected codepoint unless that codepoint is already assigned or the
codepoint is the first unallocated codepoint in the registry.

22.1.3. Reclaiming Provisional Codepoints

A request might be made to remove an unused provisional registration
from the registry to reclaim space in a registry, or portion of the
registry (such as the 64-16383 range for codepoints that use
variable-length encodings). This SHOULD be done only for the
codepoints with the earliest recorded date and entries that have been
updated less than a year prior SHOULD NOT be reclaimed.

A request to remove a codepoint MUST be reviewed by the designated
expert(s). The expert(s) MUST attempt to determine whether the
codepoint is still in use. Experts are advised to contact the listed
contacts for the registration, plus as wide a set of protocol
implementers as possible in order to determine whether any use of the
codepoint is known. The expert(s) are advised to allow at least four
weeks for responses.

If any use of the codepoints is identified by this search or a
request to update the registration is made, the codepoint MUST NOT be
reclaimed. Instead, the date on the registration is updated. A note
might be added for the registration recording relevant information
that was learned.

If no use of the codepoint was identified and no request was made to
update the registration, the codepoint MAY be removed from the
registry.

This process also applies to requests to change a provisional
registration into a permanent registration, except that the goal is
not to determine whether there is no use of the codepoint, but to
determine that the registration is an accurate representation of any
deployed usage.

22.1.4. Permanent Registrations

   Permanent registrations in QUIC registries use the Specification
   Required policy ([RFC8126]), unless otherwise specified. The
   designated expert(s) verify that a specification exists and is
   readily accessible. Expert(s) are encouraged to be biased towards
   approving registrations unless they are abusive, frivolous, or
   actively harmful (not merely aesthetically displeasing, or
   architecturally dubious). The creation of a registry MAY specify
   additional constraints on permanent registrations.

   The creation of a registry MAY identify a range of codepoints where
   registrations are governed by a different registration policy. For
   instance, the registries for 62-bit codepoints in this document have
   stricter policies for codepoints in the range from 0 to 63.

   Any stricter requirements for permanent registrations do not prevent
   provisional registrations for affected codepoints. For instance, a
   provisional registration for a frame type (Section 22.3) of 61 could
   be requested.

   All registrations made by Standards Track publications MUST be
   permanent.

   All registrations in this document are assigned a permanent status
   and list a change controller of the IETF and a contact of the QUIC
   working group (quic@ietf.org).

22.2. QUIC Transport Parameter Registry

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

   The "QUIC Transport Parameters" registry governs a 62-bit space.
   This registry follows the registration policy from Section 22.1.
   Permanent registrations in this registry are assigned using the
   Specification Required policy ([RFC8126]).

   In addition to the fields in Section 22.1.1, permanent registrations
   in this registry MUST include the following field:

   Parameter Name: A short mnemonic for the parameter.

   The initial contents of this registry are shown in Table 6.

```
+=======+======================================+===============+
| Value | Parameter Name                       | Specification |
+=======+======================================+===============+
| 0x00  | original_destination_connection_id   | Section 18.2  |
+-------+--------------------------------------+---------------+
| 0x01  | max_idle_timeout                     | Section 18.2  |
+-------+--------------------------------------+---------------+
| 0x02  | stateless_reset_token                | Section 18.2  |
+-------+--------------------------------------+---------------+
| 0x03  | max_udp_payload_size                 | Section 18.2  |
+-------+--------------------------------------+---------------+
| 0x04  | initial_max_data                     | Section 18.2  |
+-------+--------------------------------------+---------------+
| 0x05  | initial_max_stream_data_bidi_local   | Section 18.2  |
+-------+--------------------------------------+---------------+
| 0x06  | initial_max_stream_data_bidi_remote  | Section 18.2  |
+-------+--------------------------------------+---------------+
| 0x07  | initial_max_stream_data_uni          | Section 18.2  |
+-------+--------------------------------------+---------------+
| 0x08  | initial_max_streams_bidi             | Section 18.2  |
+-------+--------------------------------------+---------------+
| 0x09  | initial_max_streams_uni              | Section 18.2  |
+-------+--------------------------------------+---------------+
| 0x0a  | ack_delay_exponent                   | Section 18.2  |
+-------+--------------------------------------+---------------+
| 0x0b  | max_ack_delay                        | Section 18.2  |
+-------+--------------------------------------+---------------+
| 0x0c  | disable_active_migration             | Section 18.2  |
+-------+--------------------------------------+---------------+
| 0x0d  | preferred_address                    | Section 18.2  |
+-------+--------------------------------------+---------------+
| 0x0e  | active_connection_id_limit           | Section 18.2  |
+-------+--------------------------------------+---------------+
| 0x0f  | initial_source_connection_id         | Section 18.2  |
+-------+--------------------------------------+---------------+
| 0x10  | retry_source_connection_id           | Section 18.2  |
+-------+--------------------------------------+---------------+
```

Table 6: Initial QUIC Transport Parameters Entries

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

22.3. QUIC Frame Types Registry

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

The "QUIC Frame Types" registry governs a 62-bit space. This
registry follows the registration policy from Section 22.1.
Permanent registrations in this registry are assigned using the
Specification Required policy ([RFC8126]), except for values between
0x00 and 0x3f (in hexadecimal; inclusive), which are assigned using
Standards Action or IESG Approval as defined in Section 4.9 and 4.10
of [RFC8126].

In addition to the fields in Section 22.1.1, permanent registrations
in this registry MUST include the following field:

Frame Name: A short mnemonic for the frame type.

In addition to the advice in Section 22.1, specifications for new
permanent registrations SHOULD describe the means by which an
endpoint might determine that it can send the identified type of
frame. An accompanying transport parameter registration is expected
for most registrations; see Section 22.2. Specifications for
permanent registrations also need to describe the format and assigned
semantics of any fields in the frame.

The initial contents of this registry are tabulated in Table 3. Note
that the registry does not include the "Pkts" and "Spec" columns from
Table 3.

22.4. QUIC Transport Error Codes Registry

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

The "QUIC Transport Error Codes" registry governs a 62-bit space.
This space is split into three regions that are governed by different
policies. Permanent registrations in this registry are assigned
using the Specification Required policy ([RFC8126]), except for
values between 0x00 and 0x3f (in hexadecimal; inclusive), which are
assigned using Standards Action or IESG Approval as defined in
Section 4.9 and 4.10 of [RFC8126].

In addition to the fields in Section 22.1.1, permanent registrations
in this registry MUST include the following fields:

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.

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

Value	Error	Description	Specification
=======	=======	=============	===============
0x0	NO_ERROR	No error	Section 20
0x1	INTERNAL_ERROR	Implementation error	Section 20
0x2	CONNECTION_REFUSED	Server refuses a connection	Section 20
0x3	FLOW_CONTROL_ERROR	Flow control error	Section 20
0x4	STREAM_LIMIT_ERROR	Too many streams opened	Section 20
0x5	STREAM_STATE_ERROR	Frame received in invalid stream state	Section 20
0x6	FINAL_SIZE_ERROR	Change to final size	Section 20
0x7	FRAME_ENCODING_ERROR	Frame encoding error	Section 20
0x8	TRANSPORT_PARAMETER_ERROR	Error in transport parameters	Section 20
0x9	CONNECTION_ID_LIMIT_ERROR	Too many connection IDs received	Section 20
0xa	PROTOCOL_VIOLATION	Generic protocol violation	Section 20
0xb	INVALID_TOKEN	Invalid Token Received	Section 20
0xc	APPLICATION_ERROR	Application error	Section 20
0xd	CRYPTO_BUFFER_EXCEEDED	CRYPTO data buffer overflowed	Section 20

```
|0xe   | KEY_UPDATE_ERROR           |Invalid packet | Section 20     |
|      |                            |protection     |                |
|      |                            |update         |                |
+------+----------------------------+---------------+----------------+
|0xf   | AEAD_LIMIT_REACHED         |Excessive use of| Section 20    |
|      |                            |packet         |                |
|      |                            |protection keys|                |
+------+----------------------------+---------------+----------------+
```

           Table 7: Initial QUIC Transport Error Codes Entries

23. References

23.1. Normative References

   [DPLPMTUD] Fairhurst, G., Jones, T., Tüxen, M., Rüngeler, I., and T.
             Völker, "Packetization Layer Path MTU Discovery for
             Datagram Transports", RFC 8899, DOI 10.17487/RFC8899,
             September 2020, <https://www.rfc-editor.org/info/rfc8899>.

   [IPv4]   Postel, J., "Internet Protocol", STD 5, RFC 791,
             DOI 10.17487/RFC0791, September 1981,
             <https://www.rfc-editor.org/info/rfc791>.

   [QUIC-RECOVERY]
             Iyengar, J., Ed. and I. Swett, Ed., "QUIC Loss Detection
             and Congestion Control", Work in Progress, Internet-Draft,
             draft-ietf-quic-recovery-32, 20 October 2020,
             <https://tools.ietf.org/html/draft-ietf-quic-recovery-32>.

   [QUIC-TLS] Thomson, M., Ed. and S. Turner, Ed., "Using Transport
             Layer Security (TLS) to Secure QUIC", Work in Progress,
             Internet-Draft, draft-ietf-quic-tls-32, 20 October 2020,
             <https://tools.ietf.org/html/draft-ietf-quic-tls-32>.

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

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

   [RFC3168] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition
             of Explicit Congestion Notification (ECN) to IP",
             RFC 3168, DOI 10.17487/RFC3168, September 2001,
             <https://www.rfc-editor.org/info/rfc3168>.

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

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

   [RFC6437] Amante, S., Carpenter, B., Jiang, S., and J. Rajahalme,
              "IPv6 Flow Label Specification", RFC 6437,
              DOI 10.17487/RFC6437, November 2011,
              <https://www.rfc-editor.org/info/rfc6437>.

   [RFC8085] Eggert, L., Fairhurst, G., and G. Shepherd, "UDP Usage
              Guidelines", BCP 145, RFC 8085, DOI 10.17487/RFC8085,
              March 2017, <https://www.rfc-editor.org/info/rfc8085>.

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

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

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

   [RFC8311] Black, D., "Relaxing Restrictions on Explicit Congestion
              Notification (ECN) Experimentation", RFC 8311,
              DOI 10.17487/RFC8311, January 2018,
              <https://www.rfc-editor.org/info/rfc8311>.

   [TLS13]   Rescorla, E., "The Transport Layer Security (TLS) Protocol
              Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018,
              <https://www.rfc-editor.org/info/rfc8446>.

   [UDP]   Postel, J., "User Datagram Protocol", STD 6, RFC 768,
              DOI 10.17487/RFC0768, August 1980,
              <https://www.rfc-editor.org/info/rfc768>.

23.2. Informative References

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

   [ALPN]   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>.

   [ALTSVC]   Nottingham, M., McManus, P., and J. Reschke, "HTTP
               Alternative Services", RFC 7838, DOI 10.17487/RFC7838,
               April 2016, <https://www.rfc-editor.org/info/rfc7838>.

   [BCP38]   Ferguson, P. and D. Senie, "Network Ingress Filtering:
               Defeating Denial of Service Attacks which employ IP Source
               Address Spoofing", BCP 38, RFC 2827, DOI 10.17487/RFC2827,
               May 2000, <https://www.rfc-editor.org/info/rfc2827>.

   [COOKIE]   Barth, A., "HTTP State Management Mechanism", RFC 6265,
               DOI 10.17487/RFC6265, April 2011,
               <https://www.rfc-editor.org/info/rfc6265>.

   [CSRF]   Barth, A., Jackson, C., and J. Mitchell, "Robust defenses
               for cross-site request forgery", Proceedings of the 15th
               ACM conference on Computer and communications security -
               CCS '08, DOI 10.1145/1455770.1455782, 2008,
               <https://doi.org/10.1145/1455770.1455782>.

   [EARLY-DESIGN]
               Roskind, J., "QUIC: Multiplexed Transport Over UDP", 2
               December 2013, <https://goo.gl/dMVtFi>.

   [GATEWAY]   Hätönen, S., Nyrhinen, A., Eggert, L., Strowes, S.,
               Sarolahti, P., and M. Kojo, "An experimental study of home
               gateway characteristics", Proceedings of the 10th annual
               conference on Internet measurement - IMC '10,
               DOI 10.1145/1879141.1879174, 2010,
               <https://doi.org/10.1145/1879141.1879174>.

   [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>.

   [IPv6]   Deering, S. and R. Hinden, "Internet Protocol, Version 6
               (IPv6) Specification", STD 86, RFC 8200,
               DOI 10.17487/RFC8200, July 2017,
               <https://www.rfc-editor.org/info/rfc8200>.

   [QUIC-INVARIANTS]
              Thomson, M., "Version-Independent Properties of QUIC",
              Work in Progress, Internet-Draft, draft-ietf-quic-
              invariants-11, 20 October 2020,
              <https://tools.ietf.org/html/draft-ietf-quic-invariants-
              11>.

   [QUIC-MANAGEABILITY]
              Kuehlewind, M. and B. Trammell, "Manageability of the QUIC
              Transport Protocol", Work in Progress, Internet-Draft,
              draft-ietf-quic-manageability-07, 8 July 2020,
              <http://www.ietf.org/internet-drafts/draft-ietf-quic-
              manageability-07.txt>.

   [RFC1812] Baker, F., Ed., "Requirements for IP Version 4 Routers",
              RFC 1812, DOI 10.17487/RFC1812, June 1995,
              <https://www.rfc-editor.org/info/rfc1812>.

   [RFC1918] Rekhter, Y., Moskowitz, B., Karrenberg, D., de Groot, G.
              J., and E. Lear, "Address Allocation for Private
              Internets", BCP 5, RFC 1918, DOI 10.17487/RFC1918,
              February 1996, <https://www.rfc-editor.org/info/rfc1918>.

   [RFC2018] Mathis, M., Mahdavi, J., Floyd, S., and A. Romanow, "TCP
              Selective Acknowledgment Options", RFC 2018,
              DOI 10.17487/RFC2018, October 1996,
              <https://www.rfc-editor.org/info/rfc2018>.

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

   [RFC3449] Balakrishnan, H., Padmanabhan, V., Fairhurst, G., and M.
              Sooriyabandara, "TCP Performance Implications of Network
              Path Asymmetry", BCP 69, RFC 3449, DOI 10.17487/RFC3449,
              December 2002, <https://www.rfc-editor.org/info/rfc3449>.

   [RFC4193] Hinden, R. and B. Haberman, "Unique Local IPv6 Unicast
              Addresses", RFC 4193, DOI 10.17487/RFC4193, October 2005,
              <https://www.rfc-editor.org/info/rfc4193>.

   [RFC4291] Hinden, R. and S. Deering, "IP Version 6 Addressing
              Architecture", RFC 4291, DOI 10.17487/RFC4291, February
              2006, <https://www.rfc-editor.org/info/rfc4291>.

   [RFC4443] Conta, A., Deering, S., and M. Gupta, Ed., "Internet
              Control Message Protocol (ICMPv6) for the Internet
              Protocol Version 6 (IPv6) Specification", STD 89,
              RFC 4443, DOI 10.17487/RFC4443, March 2006,
              <https://www.rfc-editor.org/info/rfc4443>.

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

   [RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion
              Control", RFC 5681, DOI 10.17487/RFC5681, September 2009,
              <https://www.rfc-editor.org/info/rfc5681>.

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

   [RFC8087] Fairhurst, G. and M. Welzl, "The Benefits of Using
              Explicit Congestion Notification (ECN)", RFC 8087,
              DOI 10.17487/RFC8087, March 2017,
              <https://www.rfc-editor.org/info/rfc8087>.

   [SEC-CONS] Rescorla, E. and B. Korver, "Guidelines for Writing RFC
              Text on Security Considerations", BCP 72, RFC 3552,
              DOI 10.17487/RFC3552, July 2003,
              <https://www.rfc-editor.org/info/rfc3552>.

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

   [STD]   Bradner, S., "The Internet Standards Process -- Revision
              3", BCP 9, RFC 2026, DOI 10.17487/RFC2026, October 1996,
              <https://www.rfc-editor.org/info/rfc2026>.

Appendix A. Sample Packet Number Decoding Algorithm

   The pseudo-code in Figure 45 shows how an implementation can decode
   packet numbers after header protection has been removed.

```
DecodePacketNumber(largest_pn, truncated_pn, pn_nbits):
   expected_pn  = largest_pn + 1
   pn_win       = 1 << pn_nbits
   pn_hwin      = pn_win / 2
   pn_mask      = pn_win - 1
   // The incoming packet number should be greater than
   // expected_pn - pn_hwin and less than or equal to
   // expected_pn + pn_hwin
   //
   // This means we cannot just strip the trailing bits from
   // expected_pn and add the truncated_pn because that might
   // yield a value outside the window.
   //
   // The following code calculates a candidate value and
   // makes sure it's within the packet number window.
   // Note the extra checks to prevent overflow and underflow.
   candidate_pn = (expected_pn & ~pn_mask) | truncated_pn
   if candidate_pn <= expected_pn - pn_hwin and
      candidate_pn < (1 << 62) - pn_win:
      return candidate_pn + pn_win
   if candidate_pn > expected_pn + pn_hwin and
      candidate_pn >= pn_win:
      return candidate_pn - pn_win
   return candidate_pn
```

            Figure 45: Sample Packet Number Decoding Algorithm

Appendix B. Sample ECN Validation Algorithm

   Each time an endpoint commences sending on a new network path, it
   determines whether the path supports ECN; see Section 13.4. If the
   path supports ECN, the goal is to use ECN. Endpoints might also
   periodically reassess a path that was determined to not support ECN.

   This section describes one method for testing new paths. This
   algorithm is intended to show how a path might be tested for ECN
   support. Endpoints can implement different methods.

   The path is assigned an ECN state that is one of "testing",
   "unknown", "failed", or "capable". On paths with a "testing" or
   "capable" state the endpoint sends packets with an ECT marking, by
   default ECT(0); otherwise, the endpoint sends unmarked packets.

   To start testing a path, the ECN state is set to "testing" and
   existing ECN counts are remembered as a baseline.

The testing period runs for a number of packets or a limited time, as
determined by the endpoint. The goal is not to limit the duration of
the testing period, but to ensure that enough marked packets are sent
for received ECN counts to provide a clear indication of how the path
treats marked packets. Section 13.4.2 suggests limiting this to 10
packets or 3 times the probe timeout.

After the testing period ends, the ECN state for the path becomes
"unknown". From the "unknown" state, successful validation of the
ECN counts an ACK frame (see Section 13.4.2.1) causes the ECN state
for the path to become "capable", unless no marked packet has been
acknowledged.

If validation of ECN counts fails at any time, the ECN state for the
affected path becomes "failed". An endpoint can also mark the ECN
state for a path as "failed" if marked packets are all declared lost
or if they are all CE marked.

Following this algorithm ensures that ECN is rarely disabled for
paths that properly support ECN. Any path that incorrectly modifies
markings will cause ECN to be disabled. For those rare cases where
marked packets are discarded by the path, the short duration of the
testing period limits the number of losses incurred.

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-31

   * Require expansion of datagrams to ensure that a path supports at
      least 1200 bytes in both directions:

      - During the handshake ack-eliciting Initial packets from the
         server need to be expanded (#4183, #4188)

      - Path validation now requires packets containing PATH_CHALLENGE
         and PATH_RESPONSE to be expanded and PATH_RESPONSE is sent on
         the same network path (#4216, #4226)

   * Though senders need to expand datagrams in some cases, receivers
      cannot enforce this requirement (#4253, #4254)

   * Split contact into contact and change controller for IANA
      registrations (#4230, #4239)

C.2. Since draft-ietf-quic-transport-30

   * Use TRANSPORT_PARAMETER_ERROR for an invalid transport parameter
      (#4099, #4100)

   * Add a new error code for AEAD_LIMIT_REACHED code to avoid conflict
      (#4087, #4088)

   * Allow use of address validation token when server address changes
      (#4076, #4089)

C.3. Since draft-ietf-quic-transport-29

   * Require the same connection ID on coalesced packets (#3800, #3930)

   * Allow caching of packets that can't be decrypted, by allowing the
      reported acknowledgment delay to exceed max_ack_delay prior to
      confirming the handshake (#3821, #3980, #4035, #3874)

   * Allow connection ID to be used for address validation (#3834,
      #3924)

   * Required protocol operations are no longer directed at
      implementations, but are features provided to application
      protocols (#3838, #3935)

   * Narrow requirements for reset of congestion state on path change
      (#3842, #3945)

   * Add a three times amplification limit for sending of
      CONNECTION_CLOSE with reduced state (#3845, #3864)

   * Change error code for invalid RETIRE_CONNECTION_ID frames (#3860,
      #3861)

   * Recommend retention of state for lost packets to allow for late
      arrival and avoid unnecessary retransmission (#3956, #3957)

   * Allow a server to reject connections if a client reuses packet
      numbers after Retry (#3989, #3990)

   * Limit recommendation for immediate acknowledgment to when ack-
      eliciting packets are reordered (#4001, #4000)

C.4. Since draft-ietf-quic-transport-28

   * Made SERVER_BUSY error (0x2) more generic, now CONNECTION_REFUSED
      (#3709, #3690, #3694)

* Allow TRANSPORT_PARAMETER_ERROR when validating connection IDs
  (#3703, #3691)

* Integrate QUIC-specific language from draft-ietf-tsvwg-datagram-
  plpmtud (#3695, #3702)

* disable_active_migration does not apply to the addresses offered
  in server_preferred_address (#3608, #3670)

C.5. Since draft-ietf-quic-transport-27

* Allowed CONNECTION_CLOSE in any packet number space, with a
  requirement to use a new transport-level error for application-
  specific errors in Initial and Handshake packets (#3430, #3435,
  #3440)

* Clearer requirements for address validation (#2125, #3327)

* Security analysis of handshake and migration (#2143, #2387, #2925)

* The entire payload of a datagram is used when counting bytes for
  mitigating amplification attacks (#3333, #3470)

* Connection IDs can be used at any time, including in the handshake
  (#3348, #3560, #3438, #3565)

* Only one ACK should be sent for each instance of reordering
  (#3357, #3361)

* Remove text allowing a server to proceed with a bad Retry token
  (#3396, #3398)

* Ignore active_connection_id_limit with a zero-length connection ID
  (#3427, #3426)

* Require active_connection_id_limit be remembered for 0-RTT (#3423,
  #3425)

* Require ack_delay not be remembered for 0-RTT (#3433, #3545)

* Redefined max_packet_size to max_udp_datagram_size (#3471, #3473)

* Guidance on limiting outstanding attempts to retire connection IDs
  (#3489, #3509, #3557, #3547)

* Restored text on dropping bogus Version Negotiation packets
  (#3532, #3533)

* Clarified that largest acknowledged needs to be saved, but not
  necessarily signaled in all cases (#3541, #3581)

* Addressed linkability risk with the use of preferred_address
  (#3559, #3563)

* Added authentication of handshake connection IDs (#3439, #3499)

* Opening a stream in the wrong direction is an error (#3527)

C.6. Since draft-ietf-quic-transport-26

* Change format of transport parameters to use varints (#3294,
  #3169)

C.7. Since draft-ietf-quic-transport-25

* Define the use of CONNECTION_CLOSE prior to establishing
  connection state (#3269, #3297, #3292)

* Allow use of address validation tokens after client address
  changes (#3307, #3308)

* Define the timer for address validation (#2910, #3339)

C.8. Since draft-ietf-quic-transport-24

* Added HANDSHAKE_DONE to signal handshake confirmation (#2863,
  #3142, #3145)

* Add integrity check to Retry packets (#3014, #3274, #3120)

* Specify handling of reordered NEW_CONNECTION_ID frames (#3194,
  #3202)

* Require checking of sequence numbers in RETIRE_CONNECTION_ID
  (#3037, #3036)

* active_connection_id_limit is enforced (#3193, #3197, #3200,
  #3201)

* Correct overflow in packet number decode algorithm (#3187, #3188)

* Allow use of CRYPTO_BUFFER_EXCEEDED for CRYPTO frame errors
  (#3258, #3186)

* Define applicability and scope of NEW_TOKEN (#3150, #3152, #3155,
  #3156)

   * Tokens from Retry and NEW_TOKEN must be differentiated (#3127,
      #3128)

   * Allow CONNECTION_CLOSE in response to invalid token (#3168, #3107)

   * Treat an invalid CONNECTION_CLOSE as an invalid frame (#2475,
      #3230, #3231)

   * Throttle when sending CONNECTION_CLOSE after discarding state
      (#3095, #3157)

   * Application-variant of CONNECTION_CLOSE can only be sent in 0-RTT
      or 1-RTT packets (#3158, #3164)

   * Advise sending while blocked to avoid idle timeout (#2744, #3266)

   * Define error codes for invalid frames (#3027, #3042)

   * Idle timeout is symmetric (#2602, #3099)

   * Prohibit IP fragmentation (#3243, #3280)

   * Define the use of provisional registration for all registries
      (#3109, #3020, #3102, #3170)

   * Packets on one path must not adjust values for a different path
      (#2909, #3139)

C.9. Since draft-ietf-quic-transport-23

   * Allow ClientHello to span multiple packets (#2928, #3045)

   * Client Initial size constraints apply to UDP datagram payload
      (#3053, #3051)

   * Stateless reset changes (#2152, #2993)

      - tokens need to be compared in constant time

      - detection uses UDP datagrams, not packets

      - tokens cannot be reused (#2785, #2968)

   * Clearer rules for sharing of UDP ports and use of connection IDs
      when doing so (#2844, #2851)

   * A new connection ID is necessary when responding to migration
      (#2778, #2969)

   * Stronger requirements for connection ID retirement (#3046, #3096)

   * NEW_TOKEN cannot be empty (#2978, #2977)

   * PING can be sent at any encryption level (#3034, #3035)

   * CONNECTION_CLOSE is not ack-eliciting (#3097, #3098)

   * Frame encoding error conditions updated (#3027, #3042)

   * Non-ack-eliciting packets cannot be sent in response to non-ack-
      eliciting packets (#3100, #3104)

   * Servers have to change connection IDs in Retry (#2837, #3147)

C.10. Since draft-ietf-quic-transport-22

   * Rules for preventing correlation by connection ID tightened
      (#2084, #2929)

   * Clarified use of CONNECTION_CLOSE in Handshake packets (#2151,
      #2541, #2688)

   * Discourage regressions of largest acknowledged in ACK (#2205,
      #2752)

   * Improved robustness of validation process for ECN counts (#2534,
      #2752)

   * Require endpoints to ignore spurious migration attempts (#2342,
      #2893)

   * Transport parameter for disabling migration clarified to allow NAT
      rebinding (#2389, #2893)

   * Document principles for defining new error codes (#2388, #2880)

   * Reserve transport parameters for greasing (#2550, #2873)

   * A maximum ACK delay of 0 is used for handshake packet number
      spaces (#2646, #2638)

   * Improved rules for use of congestion control state on new paths
      (#2685, #2918)

   * Removed recommendation to coordinate spin for multiple connections
      that share a path (#2763, #2882)

   * Allow smaller stateless resets and recommend a smaller minimum on
      packets that might trigger a stateless reset (#2770, #2869, #2927,
      #3007).

   * Provide guidance around the interface to QUIC as used by
      application protocols (#2805, #2857)

   * Frames other than STREAM can cause STREAM_LIMIT_ERROR (#2825,
      #2826)

   * Tighter rules about processing of rejected 0-RTT packets (#2829,
      #2840, #2841)

   * Explanation of the effect of Retry on 0-RTT packets (#2842, #2852)

   * Cryptographic handshake needs to provide server transport
      parameter encryption (#2920, #2921)

   * Moved ACK generation guidance from recovery draft to transport
      draft (#1860, #2916).

C.11. Since draft-ietf-quic-transport-21

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

C.12. Since draft-ietf-quic-transport-20

   * Error codes are encoded as variable-length integers (#2672, #2680)

   * NEW_CONNECTION_ID includes a request to retire old connection IDs
      (#2645, #2769)

   * Tighter rules for generating and explicitly eliciting ACK frames
      (#2546, #2794)

   * Recommend having only one packet per encryption level in a
      datagram (#2308, #2747)

   * More normative language about use of stateless reset (#2471,
      #2574)

   * Allow reuse of stateless reset tokens (#2732, #2733)

   * Allow, but not require, enforcing non-duplicate transport
      parameters (#2689, #2691)

* Added an active_connection_id_limit transport parameter (#1994,
  #1998)

* max_ack_delay transport parameter defaults to 0 (#2638, #2646)

* When sending 0-RTT, only remembered transport parameters apply
  (#2458, #2360, #2466, #2461)

* Define handshake completion and confirmation; define clearer rules
  when it encryption keys should be discarded (#2214, #2267, #2673)

* Prohibit path migration prior to handshake confirmation (#2309,
  #2370)

* PATH_RESPONSE no longer needs to be received on the validated path
  (#2582, #2580, #2579, #2637)

* PATH_RESPONSE frames are not stored and retransmitted (#2724,
  #2729)

* Document hack for enabling routing of ICMP when doing PMTU probing
  (#1243, #2402)

C.13. Since draft-ietf-quic-transport-19

* Refine discussion of 0-RTT transport parameters (#2467, #2464)

* Fewer transport parameters need to be remembered for 0-RTT (#2624,
  #2467)

* Spin bit text incorporated (#2564)

* Close the connection when maximum stream ID in MAX_STREAMS exceeds
  2^62 - 1 (#2499, #2487)

* New connection ID required for intentional migration (#2414,
  #2413)

* Connection ID issuance can be rate-limited (#2436, #2428)

* The "QUIC bit" is ignored in Version Negotiation (#2400, #2561)

* Initial packets from clients need to be padded to 1200 unless a
  Handshake packet is sent as well (#2522, #2523)

* CRYPTO frames can be discarded if too much data is buffered
  (#1834, #2524)

   * Stateless reset uses a short header packet (#2599, #2600)

C.14. Since draft-ietf-quic-transport-18

   * Removed version negotiation; version negotiation, including
      authentication of the result, will be addressed in the next
      version of QUIC (#1773, #2313)

   * Added discussion of the use of IPv6 flow labels (#2348, #2399)

   * A connection ID can't be retired in a packet that uses that
      connection ID (#2101, #2420)

   * Idle timeout transport parameter is in milliseconds (from seconds)
      (#2453, #2454)

   * Endpoints are required to use new connection IDs when they use new
      network paths (#2413, #2414)

   * Increased the set of permissible frames in 0-RTT (#2344, #2355)

C.15. Since draft-ietf-quic-transport-17

   * Stream-related errors now use STREAM_STATE_ERROR (#2305)

   * Endpoints discard initial keys as soon as handshake keys are
      available (#1951, #2045)

   * Expanded conditions for ignoring ICMP packet too big messages
      (#2108, #2161)

   * Remove rate control from PATH_CHALLENGE/PATH_RESPONSE (#2129,
      #2241)

   * Endpoints are permitted to discard malformed initial packets
      (#2141)

   * Clarified ECN implementation and usage requirements (#2156, #2201)

   * Disable ECN count verification for packets that arrive out of
      order (#2198, #2215)

   * Use Probe Timeout (PTO) instead of RTO (#2206, #2238)

   * Loosen constraints on retransmission of ACK ranges (#2199, #2245)

   * Limit Retry and Version Negotiation to once per datagram (#2259,
      #2303)

   * Set a maximum value for max_ack_delay transport parameter (#2282,
      #2301)

   * Allow server preferred address for both IPv4 and IPv6 (#2122,
      #2296)

   * Corrected requirements for migration to a preferred address
      (#2146, #2349)

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

C.16. Since draft-ietf-quic-transport-16

   * Stream limits are defined as counts, not maximums (#1850, #1906)

   * Require amplification attack defense after closing (#1905, #1911)

   * Remove reservation of application error code 0 for STOPPING
      (#1804, #1922)

   * Renumbered frames (#1945)

   * Renumbered transport parameters (#1946)

   * Numeric transport parameters are expressed as varints (#1608,
      #1947, #1955)

   * Reorder the NEW_CONNECTION_ID frame (#1952, #1963)

   * Rework the first byte (#2006)

      - Fix the 0x40 bit

      - Change type values for long header

      - Add spin bit to short header (#631, #1988)

      - Encrypt the remainder of the first byte (#1322)

      - Move packet number length to first byte

      - Move ODCIL to first byte of retry packets

      - Simplify packet number protection (#1575)

   * Allow STOP_SENDING to open a remote bidirectional stream (#1797,
      #2013)

* Added mitigation for off-path migration attacks (#1278, #1749, #2033)

* Don't let the PMTU to drop below 1280 (#2063, #2069)

* Require peers to replace retired connection IDs (#2085)

* Servers are required to ignore Version Negotiation packets (#2088)

* Tokens are repeated in all Initial packets (#2089)

* Clarified how PING frames are sent after loss (#2094)

* Initial keys are discarded once Handshake are available (#1951, #2045)

* ICMP PTB validation clarifications (#2161, #2109, #2108)

C.17. Since draft-ietf-quic-transport-15

   Substantial editorial reorganization; no technical changes.

C.18. Since draft-ietf-quic-transport-14

* Merge ACK and ACK_ECN (#1778, #1801)

* Explicitly communicate max_ack_delay (#981, #1781)

* Validate original connection ID after Retry packets (#1710, #1486, #1793)

* Idle timeout is optional and has no specified maximum (#1765)

* Update connection ID handling; add RETIRE_CONNECTION_ID type (#1464, #1468, #1483, #1484, #1486, #1495, #1729, #1742, #1799, #1821)

* Include a Token in all Initial packets (#1649, #1794)

* Prevent handshake deadlock (#1764, #1824)

C.19. Since draft-ietf-quic-transport-13

* Streams open when higher-numbered streams of the same type open (#1342, #1549)

* Split initial stream flow control limit into 3 transport parameters (#1016, #1542)

   * All flow control transport parameters are optional (#1610)

   * Removed UNSOLICITED_PATH_RESPONSE error code (#1265, #1539)

   * Permit stateless reset in response to any packet (#1348, #1553)

   * Recommended defense against stateless reset spoofing (#1386,
      #1554)

   * Prevent infinite stateless reset exchanges (#1443, #1627)

   * Forbid processing of the same packet number twice (#1405, #1624)

   * Added a packet number decoding example (#1493)

   * More precisely define idle timeout (#1429, #1614, #1652)

   * Corrected format of Retry packet and prevented looping (#1492,
      #1451, #1448, #1498)

   * Permit 0-RTT after receiving Version Negotiation or Retry (#1507,
      #1514, #1621)

   * Permit Retry in response to 0-RTT (#1547, #1552)

   * Looser verification of ECN counters to account for ACK loss
      (#1555, #1481, #1565)

   * Remove frame type field from APPLICATION_CLOSE (#1508, #1528)

C.20. Since draft-ietf-quic-transport-12

   * Changes to integration of the TLS handshake (#829, #1018, #1094,
      #1165, #1190, #1233, #1242, #1252, #1450, #1458)

      - The cryptographic handshake uses CRYPTO frames, not stream 0

      - QUIC packet protection is used in place of TLS record
         protection

      - Separate QUIC packet number spaces are used for the handshake

      - Changed Retry to be independent of the cryptographic handshake

      - Added NEW_TOKEN frame and Token fields to Initial packet

      - Limit the use of HelloRetryRequest to address TLS needs (like
         key shares)

* Enable server to transition connections to a preferred address
  (#560, #1251, #1373)

* Added ECN feedback mechanisms and handling; new ACK_ECN frame
  (#804, #805, #1372)

* Changed rules and recommendations for use of new connection IDs
  (#1258, #1264, #1276, #1280, #1419, #1452, #1453, #1465)

* Added a transport parameter to disable intentional connection
  migration (#1271, #1447)

* Packets from different connection ID can't be coalesced (#1287,
  #1423)

* Fixed sampling method for packet number encryption; the length
  field in long headers includes the packet number field in addition
  to the packet payload (#1387, #1389)

* Stateless Reset is now symmetric and subject to size constraints
  (#466, #1346)

* Added frame type extension mechanism (#58, #1473)

C.21. Since draft-ietf-quic-transport-11

* Enable server to transition connections to a preferred address
  (#560, #1251)

* Packet numbers are encrypted (#1174, #1043, #1048, #1034, #850,
  #990, #734, #1317, #1267, #1079)

* Packet numbers use a variable-length encoding (#989, #1334)

* STREAM frames can now be empty (#1350)

C.22. 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.23. 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.24. 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.25. Since draft-ietf-quic-transport-07

   * The long header now has version before packet number (#926, #939)

   * Rename and consolidate packet types (#846, #822, #847)

   * Packet types are assigned new codepoints and the Connection ID
      Flag is inverted (#426, #956)

   * Removed type for Version Negotiation and use Version 0 (#963,
      #968)

   * Streams are split into unidirectional and bidirectional (#643,
      #656, #720, #872, #175, #885)

- Stream limits now have separate uni- and bi-directional
  transport parameters (#909, #958)

- Stream limit transport parameters are now optional and default
  to 0 (#970, #971)

* The stream state machine has been split into read and write (#634,
  #894)

* Employ variable-length integer encodings throughout (#595)

* Improvements to connection close

  - Added distinct closing and draining states (#899, #871)

  - Draining period can terminate early (#869, #870)

  - Clarifications about stateless reset (#889, #890)

* Address validation for connection migration (#161, #732, #878)

* Clearly defined retransmission rules for BLOCKED (#452, #65, #924)

* negotiated_version is sent in server transport parameters (#710,
  #959)

* Increased the range over which packet numbers are randomized
  (#864, #850, #964)

C.26. 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.27. 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.28. Since draft-ietf-quic-transport-04

   * Introduce STOP_SENDING frame, RESET_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.29. Since draft-ietf-quic-transport-03

   * Change STREAM and RESET_STREAM layout

   * Add MAX_STREAM_ID settings

C.30. 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.31. 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 RESET_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)

   * RESET_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 RESET_STREAM and CONNECTION_CLOSE are now at a
      consistent offset (#249)

   * Defined priority as the responsibility of the application protocol
      (#104, #303)

C.32. 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.33. 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

Contributors

   The original design and rationale behind this protocol draw
   significantly from work by Jim Roskind [EARLY-DESIGN].

   The IETF QUIC Working Group received an enormous amount of support
   from many people. The following people provided substantive
   contributions to this document:

   * Alessandro Ghedini

   * Alyssa Wilk

   * Antoine Delignat-Lavaud

   * Brian Trammell

   * Christian Huitema

   * Colin Perkins

   * David Schinazi

   * Dmitri Tikhonov

   * Eric Kinnear

   * Eric Rescorla

   * Gorry Fairhurst

   * Ian Swett

   * Igor Lubashev

   * å¥¥ ä¸ç© (Kazuho Oku)

   * Lars Eggert

   * Lucas Pardue

   * Magnus Westerlund

   * Marten Seemann

   * Martin Duke

   * Mike Bishop

   * Mikkel FahnÃ¸e JÃ¸rgensen

   * Mirja KÃ¼hlewind

   * Nick Banks

   * Nick Harper

   * Patrick McManus

   * Roberto Peon

   * Ryan Hamilton

   * Subodh Iyengar

   * Tatsuhiro Tsujikawa

   * Ted Hardie

   * Tom Jones

   * Victor Vasiliev

Authors' Addresses

   Jana Iyengar (editor)
   Fastly

      Email: jri.ietf@gmail.com

   Martin Thomson (editor)
   Mozilla

      Email: mt@lowentropy.net