Network Working Group
Internet-Draft
Intended status: Experimental
Expires: May 4, 2017

R. Hamilton
J. Iyengar
I. Swett
A. Wilk
Google

October 31, 2016

QUIC: A UDP-Based Multiplexed and Secure Transport
draft-hamilton-quic-transport-protocol-01

Abstract

QUIC is a multiplexed and secure transport protocol that runs on top of UDP. QUIC builds on past transport experience, and implements mechanisms that make it useful as a modern general-purpose transport protocol. Using UDP as the basis of QUIC is intended to address compatibility issues with legacy clients and middleboxes. QUIC authenticates all of its headers, preventing third parties from from changing them. QUIC encrypts most of its headers, thereby limiting protocol evolution to QUIC endpoints only. Therefore, middleboxes, in large part, are not required to be updated as new protocol versions are deployed. This document describes the core QUIC protocol, including the conceptual design, wire format, and mechanisms of the QUIC protocol for connection establishment, stream multiplexing, stream and connection-level flow control, and data reliability. Accompanying documents describe QUIC's loss recovery and congestion control, and the use of TLS 1.3 for key negotiation.

Status of This Memo

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

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

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

This Internet-Draft will expire on May 4, 2017.

Table of Contents

1. Introduction

QUIC is a multiplexed and secure transport protocol that runs on top of UDP. QUIC builds on past transport experience and implements mechanisms that make it useful as a modern general-purpose transport protocol. Using UDP as the substrate, QUIC seeks to be compatible with legacy clients and middleboxes. QUIC authenticates all of its headers, preventing middleboxes and other third parties from changing them, and encrypts most of its headers, limiting protocol evolution largely to QUIC endpoints only.

This document describes the core QUIC protocol, including the conceptual design, wire format, and mechanisms of the QUIC protocol

for connection establishment, stream multiplexing, stream and connection-level flow control, and data reliability. Accompanying documents describe QUIC's loss detection and congestion control [draft-iyengar-quic-loss-detection], and the use of TLS 1.3 for key negotiation [draft-thomson-quic-tls].

2. Conventions and Definitions

Definitions of terms that are used in this document:

- Client: The endpoint initiating a QUIC connection.
- Server: The endpoint accepting incoming QUIC connections.
- Endpoint: The client or server end of a connection.
- Stream: A logical, bi-directional channel of ordered bytes within a QUIC connection.
- Connection: A conversation between two QUIC endpoints with a single encryption context that multiplexes streams within it.
- Connection ID: The identifier for a QUIC connection.
- QUIC packet: A well-formed UDP payload that can be parsed by a QUIC receiver. QUIC packet size in this document refers to the UDP payload size.

3. A QUIC Overview

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

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

3.1. Low-Latency Version Negotiation

QUIC combines version negotiation with the rest of connection establishment to avoid unnecessary roundtrip delays. A QUIC client proposes a version to use for the connection, and encodes the rest of the handshake using the proposed version. If the server does not speak the client-chosen version, it forces version negotiation by sending back a Version Negotiation packet to the client, causing a roundtrip of delay before connection establishment.

This mechanism eliminates roundtrip latency when the client's optimistically-chosen version is spoken by the server, and incentivizes servers to not lag behind clients in deployment of newer versions. Additionally, an application may negotiate QUIC versions out-of-band to increase chances of success in the first roundtrip and to obviate the additional roundtrip in the case of version mismatch.

3.2. Low-Latency Connection Establishment

QUIC relies on a combined crypto and transport handshake for setting up a secure transport connection. QUIC connections are expected to commonly use 0-RTT handshakes, meaning that for most QUIC connections, data can be sent immediately following the client handshake packet, without waiting for a reply from the server. QUIC provides a dedicated stream (Stream ID 1) to be used for performing the crypto handshake and QUIC options negotiation. The format of the QUIC options and parameters used during negotiation are described in this document, but the handshake protocol that runs on Stream ID 1 is described in the accompanying crypto handshake draft [draft-thomson-quic-tls].

3.3. Stream Multiplexing

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

3.4. Rich Signaling for Congestion Control and Loss Recovery

QUIC's packet framing and acknowledgments carry rich information that help both congestion control and loss recovery in fundamental ways. Each QUIC packet carries a new packet number, including those

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

3.5. Stream and Connection Flow Control

QUIC implements stream- and connection-level flow control, closely following HTTP/2's flow control mechanisms. At a high level, a QUIC receiver advertises the absolute byte offset within each stream up to which the receiver is willing to receive data. As data is sent, received, and delivered on a particular stream, the receiver sends WINDOW_UPDATE frames that increase the advertised offset limit for that stream, allowing the peer to send more data on that stream. In addition to this stream-level flow control, QUIC implements connection-level flow control to limit the aggregate buffer that a QUIC receiver is willing to allocate to all streams on a connection. Connection-level flow control works in the same way as stream-level flow control, but the bytes delivered and highest received offset are all aggregates across all streams.

3.6. Authenticated and Encrypted Header and Payload

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

Generally, QUIC packets are always authenticated and the payload is typically fully encrypted. The parts of the packet header which are not encrypted are still authenticated by the receiver, so as to thwart any packet injection or manipulation by third parties. Some early handshake packets, such as the Version Negotiation packet, are not encrypted, but information sent in these unencrypted handshake packets is later verified under crypto cover.

PUBLIC_RESET packets that reset a connection are currently not authenticated.

3.7. Connection Migration and Resilience to NAT Rebinding

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

4. Packet Types and Formats

We first describe QUIC's packet types and formats, since some are referenced in subsequent mechanisms. Note that unless otherwise noted, all values specified in this document are in little-endian format and all field sizes are in bits.

4.1. Common Header

All QUIC packets begin with a QUIC Common header, as shown below:

The fields in the Common Header are the following:

- Flags:
  - 0x01 = VERSION. The semantics of this flag depends on whether the packet is sent by the server or the client. A client MAY set this flag and include exactly one proposed version. A server may set this flag when the client-proposed version was unsupported, and may then provide a list (0 or more) of acceptable versions as a part of version negotiation (described in Section XXX.)
  - 0x02 = PUBLIC_RESET. Set to indicate that the packet is a Public Reset packet.

  * 0x04 = DIVERSIFICATION_NONCE. Set to indicate the presence of a 32-byte diversification nonce in the header. (DISCUSS_AND_MODIFY: This flag should be removed along with the Diversification Nonce bits, as discussed further below.)

  * 0x08 = CONNECTION_ID. Indicates the Connection ID is present in the packet. This must be set in all packets until negotiated to a different value for a given direction. For instance, if a client indicates that the 5-tuple fully identifies the connection at the client, the connection ID is optional in the server-to-client direction.

  * 0x30 = PACKET_NUMBER_SIZE. These two bits indicate the number of low-order-bytes of the packet number that are present in each packet.

    + 11 indicates that 6 bytes of the packet number are present

    + 10 indicates that 4 bytes of the packet number are present

    + 01 indicates that 2 bytes of the packet number are present

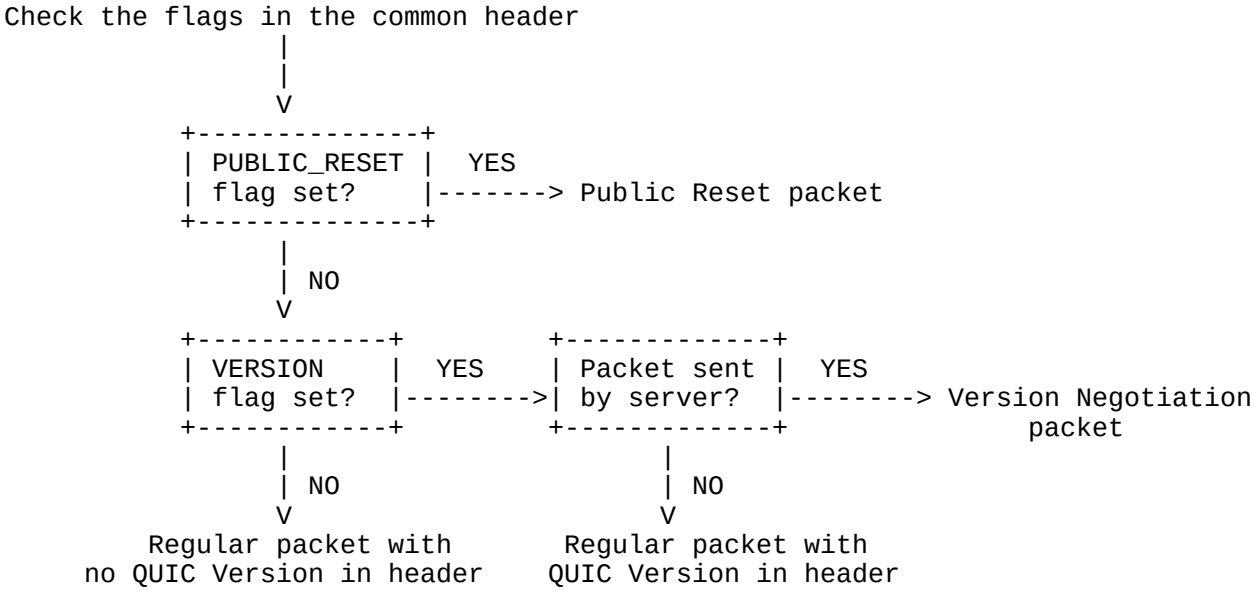
    + 00 indicates that 1 byte of the packet number is present

  * 0x40 = MULTIPATH. This bit is reserved for multipath use.

  * 0x80 is currently unused, and must be set to 0.

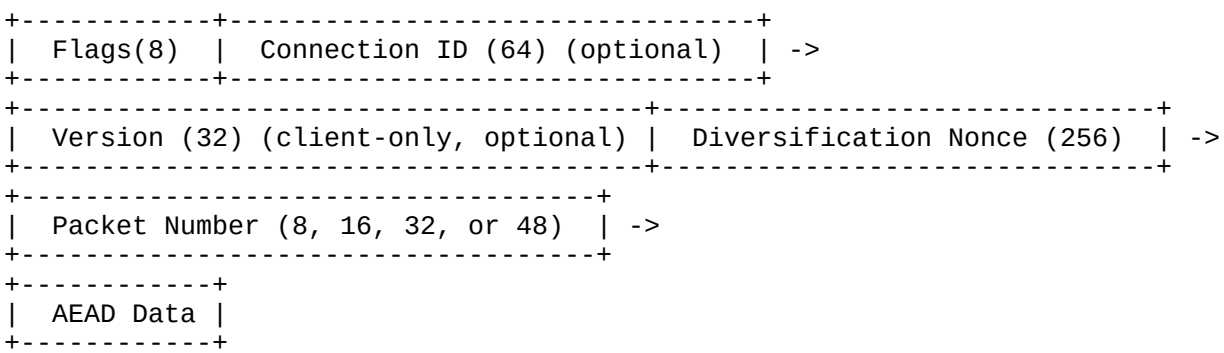
- Connection ID: An unsigned 64-bit random number chosen by the client, used as the identifier of the connection. Connection ID is tied to a QUIC connection, and remains consistent across client and/or server IP and port changes.

While all QUIC packets have the same common header, there are three types of packets: Regular packets, Version Negotiation packets, and Public Reset packets. The flowchart below shows how a packet is classified into one of these three packet types:



4.2. Regular Packets

Each Regular packet's header consists of a Common Header followed by fields specific to Regular packets, as shown below:



The fields in a Regular packet past the Common Header are the following:

- QUIC Version: A 32-bit opaque tag that represents the version of the QUIC protocol. Only present in the client-to-server direction, and if the VERSION flag is set. Version Negotiation is described in Section XXX.

- DISCUSS_AND_REPLACE: Diversification Nonce: A 32-byte nonce generated by the server and used only in the Server->Client direction to ensure that the server is able to generate unique keys per connection. Specifically, when using QUIC's 0-RTT crypto handshake, a repeated CHLO with the exact same connection ID and CHLO can lead to the same (intermediate) initial-encryption keys being derived for the connection. A server-generated nonce disallows a client from causing the same keys to be derived for two distinct connections. Once the connection is forward-secure, this nonce is no longer present in packets. This nonce can be removed from the packet header if a requirement can be added for the crypto handshake to ensure key uniqueness. The expectation is that TLS1.3 meets this requirement. Upon working group adoption of this document, this requirement should be added to the crypto handshake requirements, and the nonce should be removed from the packet format.

- Packet Number: The lower 8, 16, 32, or 48 bits of the packet number, based on the PACKET_NUMBER_SIZE flag. Each Regular packet is assigned a packet number by the sender. The first packet sent by an endpoint MUST have a packet number of 1.

- AEAD Data: A Regular packet's header, which includes the Common Header, and the Version, Diversification Nonce, and Packet Number fields, is authenticated but not encrypted. The rest of a Regular packet, starting with the first frame, is both authenticated and encrypted. Immediately following the header, Regular packets contain AEAD (Authenticated Encryption with Associated Data) data. This data must be decrypted in order for the contents to be interpreted. After decryption, the plaintext consists of a sequence of frames, as shown (frames are described in Section XXX).

4.2.1. Packet Number Compression and Reconstruction

The complete packet number is a 64-bit unsigned number and is used as part of a cryptographic nonce for packet encryption. To reduce the number of bits required to represent the packet number over the wire, at most 48 bits of the packet number are transmitted over the wire. A QUIC endpoint MUST NOT reuse a complete packet number within the same connection (that is, under the same cryptographic keys). If the total number of packets transmitted in this connection reaches 2^64 - 1, the sender MUST close the connection by sending a CONNECTION_CLOSE

frame with the error code QUIC_SEQUENCE_NUMBER_LIMIT_REACHED (connection termination is described in Section XXX.) For unambiguous reconstruction of the complete packet number by a receiver from the lower-order bits, a QUIC sender MUST NOT have more than 2^(packet_number_size - 2) in flight at any point in the connection. In other words,

- If a sender sets PACKET_NUMBER_SIZE bits to 11, it MUST NOT have more than (2^46) packets in flight.
- If a sender sets PACKET_NUMBER_SIZE bits to 10, it MUST NOT have more than (2^30) packets in flight.
- If a sender sets PACKET_NUMBER_SIZE bits to 01, it MUST NOT have more than (2^14) packets in flight.
- If a sender sets PACKET_NUMBER_SIZE bits to 00, it MUST NOT have more than (2^6) packets in flight.

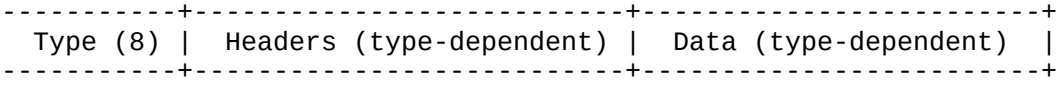
DISCUSS: Should the receiver be required to enforce this rule that the sender MUST NOT exceed the inflight limit? Specifically, should the receiver drop packets that are received outside this window?

Any truncated packet number received from a peer MUST be reconstructed as the value closest to the next expected packet number from that peer.

(TODO: Clarify how packet number size can change mid-connection.)

4.2.2. Frames and Frame Types

A Regular packet MUST contain at least one frame, and MAY contain multiple frames and multiple frame types. Frames MUST fit within a single QUIC packet and MUST NOT span a QUIC packet boundary. Each frame begins with a Frame Type byte, indicating its type, followed by type-dependent headers, and variable-length data, as follows:

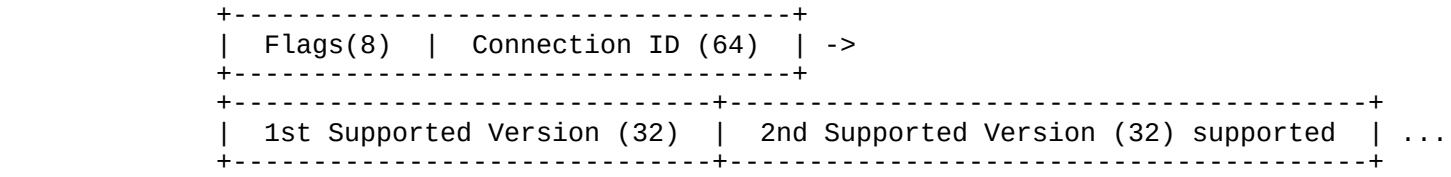


The following table lists currently defined frame types. Note that the Frame Type byte in STREAM and ACK frames is used to carry other frame-specific flags. For all other frames, the Frame Type byte simply identifies the frame. These frames are explained in more detail as they are referenced later in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

Type-field value	Frame type
1FDOOOSS	STREAM
01NTLLMM	ACK
00000000 (0x00)	PADDING
00000001 (0x01)	RST_STREAM
00000010 (0x02)	CONNECTION_CLOSE
00000011 (0x03)	GOAWAY
00000100 (0x04)	WINDOW_UPDATE
00000101 (0x05)	BLOCKED
00000110 (0x06)	STOP_WAITING
00000111 (0x07)	PING

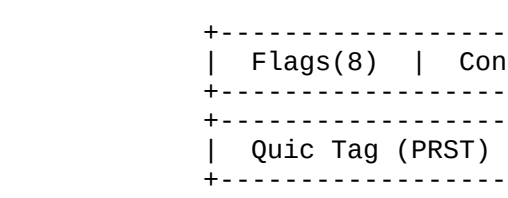
4.3. Version Negotiation Packet

A Version Negotiation packet is only sent by the server, MUST have the VERSION flag set, and MUST include the full 64-bit Connection ID. The rest of the Version Negotiation packet is a list of 4-byte versions which the server supports, as shown below.



4.4. Public Reset Packet

A Public Reset packet MUST have the PUBLIC_RESET flag set, and MUST include the full 64-bit connection ID. The rest of the Public Reset packet is encoded as if it were a crypto handshake message of the tag PRST, as shown below.



The tag value map contains the following tag-values:

- RNON (public reset nonce proof) - a 64-bit unsigned integer.
- RSEQ (rejected packet number) - a 64-bit packet number.

- CADR (client address) - the observed client IP address and port number. This is currently for debugging purposes only and hence is optional.

DISCUSS_AND_REPLACE: The crypto handshake message format is described in the QUIC crypto document, and should be replaced with something simpler when this document is adopted. The purpose of the tag-value map following the PRST tag is to enable the receiver of the Public Reset packet to reasonably authenticate the packet. This map is an extensible map format that allows specification of various tags, which should again be replaced by something simpler.

5. Life of a Connection

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

5.1. Version Negotiation

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

A QUIC connection begins with a client sending a handshake packet. The details of the handshake mechanisms are described in Section XX, but all of the initial packets sent from the client to the server MUST have the VERSION flag set, and MUST specify the version of the protocol being used.

When the server receives a packet from a client with the VERSION flag set for a connection that has not yet been established, it compares the client's version to the versions it supports.

- If the client's version is acceptable to the server, the server MUST use this protocol version for the lifetime of the connection. All subsequent packets sent by the server MUST have the version flag off.

- If the client's version is not acceptable to the server, the server MUST send a Version Negotiation packet to the client. This packet will have the VERSION flag set and will include the server's set of supported versions. On subsequently received

packets for the same connection ID with the unacceptable version, the server MUST continue responding with a Version Negotiation packet.

When the client receives a Version Negotiation packet from the server, it should select an acceptable protocol version. If such a version is found, the client MUST resend all packets using the new version, and the resent packets MUST use new packet numbers. These packets MUST continue to have the VERSION flag set and MUST include the new negotiated protocol version.

The client MUST send its version on all packets until it receives a packet from the server with the VERSION flag off. If version negotiation is successful, the client should receive a packet from the server with the VERSION flag off indicating the end of version negotiation. All subsequent packets the client sends MUST have the version flag off.

Once the server receives a packet from the client with the VERSION flag off, it MUST ignore the VERSION flag in subsequently received packets.

The Version Negotiation packet is unencrypted and exchanged without authentication. To avoid a downgrade attack, the client needs to verify its record of the server's version list in the Version Negotiation packet and the server needs to verify its record of the client's originally proposed version. Therefore, the client and server MUST include this information later in their corresponding crypto handshake data.

5.2. Crypto and Transport Handshake

QUIC relies on a combined crypto and transport handshake to minimize connection establishment latency. QUIC provides a dedicated stream (Stream ID 1) to be used for performing a combined connection and security handshake (streams are described in detail in Section XXX). The crypto handshake protocol encapsulates and delivers QUIC's transport handshake to the peer on the crypto stream. The first QUIC packet from the client to the server MUST carry handshake information as data on Stream ID 1.

5.2.1. Transport Parameters and Options

During connection establishment, the handshake must negotiate various transport parameters. The currently defined transport parameters are described later in the document.

The transport component of the handshake is responsible for exchanging and negotiating the following parameters for a QUIC connection. Not all parameters are negotiated, some are parameters sent in just one direction. These parameters and options are encoded and handed off to the crypto handshake protocol to be transmitted to the peer.

5.2.1.1. Encoding

(TODO: Describe format with example)

QUIC encodes the transport parameters and options as tag-value pairs, all as 7-bit ASCII strings. QUIC parameter tags are listed below.

5.2.1.2. Required Transport Parameters

- SFCW: Stream Flow Control Window. The stream level flow control byte offset advertised by the sender of this parameter.
- CFCW: Connection Flow Control Window. The connection level flow control byte offset advertised by the sender of this parameter.
- MSPC: Maximum number of incoming streams per connection.

5.2.1.3. Optional Transport Parameters

- TCID: Indicates support for truncated Connection IDs. If sent by a peer, indicates that connection IDs sent to the peer should be truncated to 0 bytes. This is expected to commonly be used by an endpoint where the 5-tuple is sufficient to identify a connection. For instance, if the 5-tuple is unique at the client, the client MAY send a TCID parameter to the server. When a TCID parameter is received, an endpoint MAY choose to not send the connection ID on subsequent packets.
- COPT: Connection Options are a repeated tag field. The field contains any connection options being requested by the client or server. These are typically used for experimentation and will evolve over time. Example use cases include changing congestion control algorithms and parameters such as initial window. (TODO: List connection options.)

5.2.2. Proof of Source Address Ownership

Transport protocols commonly use a roundtrip time to verify a client’s address ownership for protection from malicious clients that spoof their source address. QUIC uses a cookie, called the Source Address Token (STK), to mostly eliminate this roundtrip of delay.

This technique is similar to TCP Fast Open's use of a cookie to avoid a roundtrip of delay in TCP connection establishment.

On a new connection, a QUIC server sends an STK, which is opaque to and stored by the client. On a subsequent connection, the client echoes it in the transport handshake as proof of IP ownership.

A QUIC server also uses the STK to store server-designated connection IDs for Stateless Rejects, to verify that an incoming connection contains the correct connection ID.

A QUIC server MAY additionally store other data in a the STK, such as measured bandwidth and measured minimum RTT to the client that may help the server better bootstrap a subsequent connection from the same client. A server MAY send an updated STK message mid-connection to update server state that is stored at the client in the STK.

(TODO: Describe server and client actions on STK, encoding, recommendations for what to put in an STK. Describe SCUP messages.)

5.2.3. Crypto Handshake Protocol Features

QUIC's current crypto handshake mechanism is documented in [QUIC-CRYPTO]. QUIC does not restrict itself to using a specific handshake protocol, so the details of a specific handshake protocol are out of this document's scope. If not explicitly specified in the application mapping, TLS is assumed to be the default crypto handshake protocol, as described in [draft-mthomson-quic-tls]. An application that maps to QUIC MAY however specify an alternative crypto handshake protocol to be used.

The following list of requirements and recommendations documents properties of the current prototype handshake which should be provided by any handshake protocol.

- The crypto handshake MUST ensure that the final negotiated key is distinct for every connection between two endpoints.

- Transport Negotiation: The crypto handshake MUST provide a mechanism for the transport component to exchange transport parameters and Source Address Tokens. To avoid downgrade attacks, the transport parameters sent and received MUST be verified before the handshake completes successfully.

- Connection Establishment in 0-RTT: Since low-latency connection establishment is a critical feature of QUIC, the QUIC handshake protocol SHOULD attempt to achieve 0-RTT connection establishment latency for repeated connections between the same endpoints.

- Source Address Spoofing Defense: Since QUIC handles source address verification, the crypto protocol SHOULD NOT impose a separate source address verification mechanism.

- Server Config Update: A QUIC server may refresh the source-address token (STK) mid-connection, to update the information stored in the STK at the client and to extend the period over which 0-RTT connections can be established by the client.

- Certificate Compression: Early QUIC experience demonstrated that compressing certificates exchanged during a handshake is valuable in reducing latency. This additionally helps to reduce the amplification attack footprint when a server sends a large set of certificates, which is not uncommon with TLS. The crypto protocol SHOULD compress certificates and any other information to minimize the number of packets sent during a handshake.

The following information used during the QUIC handshake MUST be cryptographically verified by the crypto handshake protocol:

- Client's originally proposed version in its first packet.

- Server's version list in it's Version Negotiation packet, if one was sent.

5.3. Connection Migration

QUIC connections are identified by their 64-bit Connection ID. QUIC's consistent connection ID allows connections to survive changes to the client's IP and/or port, such as those caused by client or server migrating to a new network. QUIC also provides automatic cryptographic verification of a rebound client, since the client continues to use the same session key for encrypting and decrypting packets.

DISCUSS: Simultaneous migration. Is this reasonable?

TODO: Perhaps move mitigation techniques from Security Considerations here.

5.4. Connection Termination

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

1. Explicit Shutdown: An endpoint sends a CONNECTION_CLOSE frame to the peer initiating a connection termination. An endpoint may

send a GOAWAY frame to the peer prior to a CONNECTION_CLOSE to indicate that the connection will soon be terminated. A GOAWAY frame signals to the peer that any active streams will continue to be processed, but the sender of the GOAWAY will not initiate any additional streams and will not accept any new incoming streams. On termination of the active streams, a CONNECTION_CLOSE may be sent. If an endpoint sends a CONNECTION_CLOSE frame while unterminated streams are active (no FIN bit or RST_STREAM frames have been sent or received for one or more streams), then the peer must assume that the streams were incomplete and were abnormally terminated.

2. Implicit Shutdown: The default idle timeout for a QUIC connection is 30 seconds, and is a required parameter (ICSL) in connection negotiation. The maximum is 10 minutes. If there is no network activity for the duration of the idle timeout, the connection is closed. By default a CONNECTION_CLOSE frame will be sent. A silent close option can be enabled when it is expensive to send an explicit close, such as mobile networks that must wake up the radio.

3. Abrupt Shutdown: An endpoint may send a Public Reset packet at any time during the connection to abruptly terminate an active connection. A Public Reset packet SHOULD only be used as a final recourse. Commonly, a public reset is expected to be sent when a packet on an established connection is received by an endpoint that is unable decrypt the packet. For instance, if a server reboots mid-connection and loses any cryptographic state associated with open connections, and then receives a packet on an open connection, it should send a Public Reset packet in return. (TODO: articulate rules around when a public reset should be sent.)

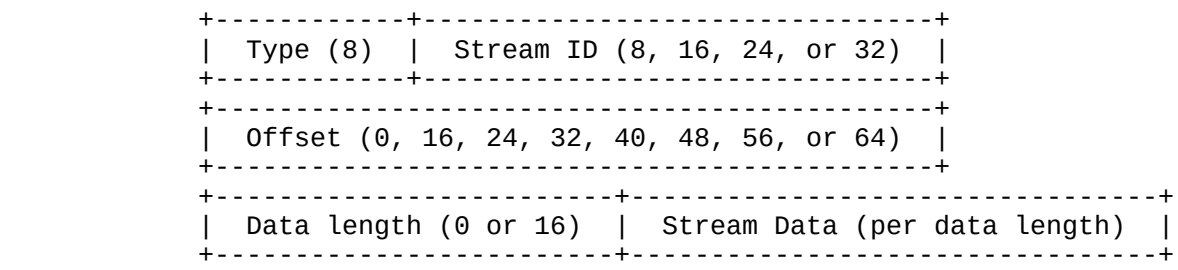
TODO: Connections that are terminated are added to a TIME_WAIT list at the server, so as to absorb any straggler packets in the network. Discuss TIME_WAIT list.

6. Frame Types and Formats

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

6.1. STREAM Frame

STREAM frames implicitly create a stream and carry stream data. A STREAM frame is shown below.



The STREAM frame header fields are as follows:

- Frame Type: The Frame Type byte is an 8-bit value containing various flags, and is formatted as the following 8 bits: 1FDOOOSS.
  - The leftmost bit must be set to 1 indicating that this is a STREAM frame.
  - 'F' is the FIN bit, which is used for stream termination.
  - The 'D' bit indicates whether a Data Length field is present in the STREAM header. When set to 0, this field indicates that the Stream Data field extends to the end of the packet. When set to 1, this field indicates that Data Length field contains the length (in bytes) of the Stream Data field. The option to omit the length should only be used when the packet is a "full-sized" packet, to avoid the risk of corruption via padding.
  - The 'OOO' bits encode the length of the Offset header field as 0, 16, 24, 32, 40, 48, 56, or 64 bits long.
  - The 'SS' bits encode the length of the Stream ID header field as 8, 16, 24, or 32 bits. (DISCUSS: Consider making this 8, 16, 32, 64.)
- Stream ID: A variable-sized unsigned ID unique to this stream.
- Offset: A variable-sized unsigned number specifying the byte offset in the stream for the data in this STREAM frame. The first byte in the stream has an offset of 0.
- Data Length: An optional 16-bit unsigned number specifying the length of the Stream Data field in this STREAM frame.

A STREAM frame MUST have either non-zero data length or the FIN bit set.

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

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

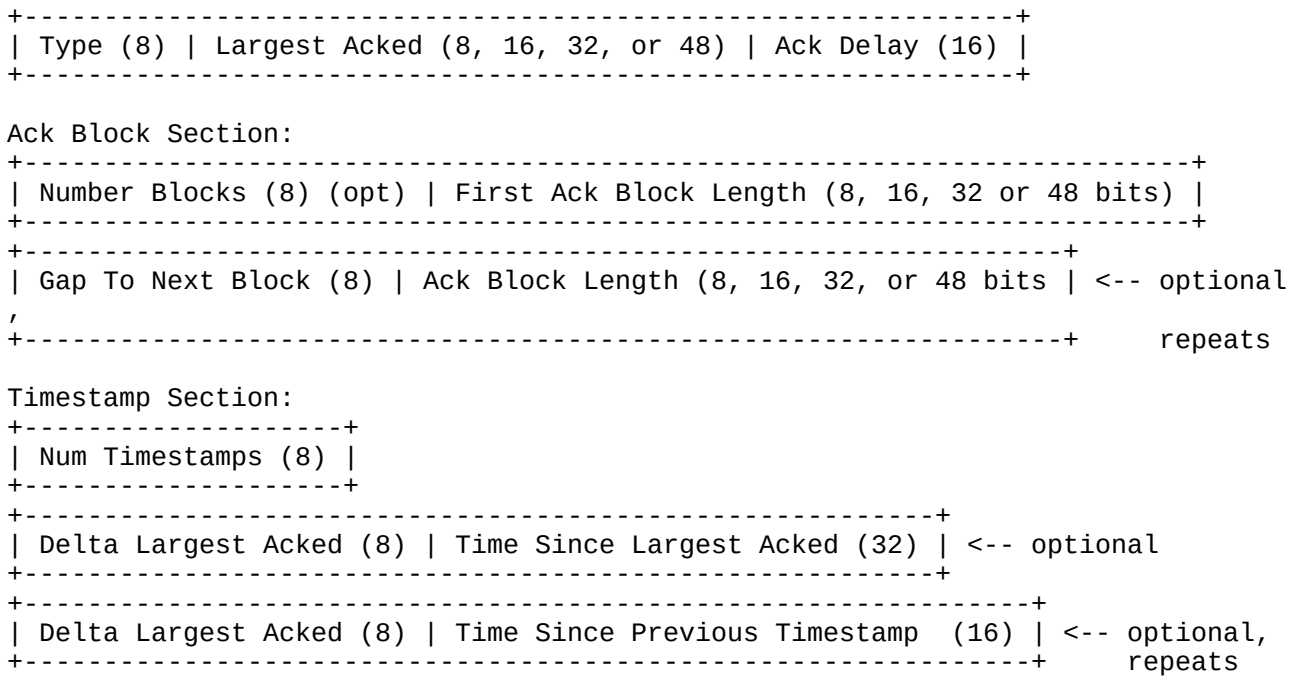
6.2. ACK Frame

Receivers send ACK frames to inform senders which packets they have received, as well as which packets are considered missing. The ACK frame contains between 1 and 256 ack blocks. Ack blocks are ranges of acknowledged packets.

To limit the ACK blocks to the ones that haven’t yet been received by the sender, the sender periodically sends STOP_WAITING frames that signal the receiver to stop acking packets below a specified sequence number, raising the "least unacked" packet number at the receiver. A sender of an ACK frame thus reports only those ACK blocks between the received least unacked and the reported largest observed packet numbers. It is recommended for the sender to send the most recent largest acked packet it has received in an ack as the STOP_WAITING frame’s least unacked value.

Unlike TCP SACKs, QUIC ACK blocks are irrevocable. Once a packet is acked, even if it does not appear in a future ack frame, it is assumed to be acked.

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



The fields in the ACK frame are as follows:

- Frame Type: The Frame Type byte is an 8-bit value containing various flags. This byte is formatted as the following 8 bits: 01NULLMM.
  - The first two bits must be set to 01 indicating that this is an ACK frame.
  - The 'N' bit indicates whether the frame has more than 1 ack range.
  - The 'U' bit is unused.
  - The two 'LL' bits encode the length of the Largest Acked field as 1, 2, 4, or 6 bytes long.
  - The two 'MM' bits encode the length of the Ack Block Length fields as 1, 2, 4, or 6 bytes long.
- Largest Acked: A variable-sized unsigned value representing the largest packet number the peer is acking in this packet (typically the largest that the peer has seen thus far.)

- Ack Delay: Time from when the largest acked, as indicated in the Largest Acked field, was received by this peer to when this ack was sent.

- Ack Block Section:

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

  * First Ack Block Length: An unsigned packet number delta that indicates the number of contiguous additional packets being acked starting at the Largest Acked.

  * Gap To Next Block (opt, repeated): An unsigned number specifying the number of contiguous missing packets from the end of the previous ack block to the start of the next.

  * Ack Block Length (opt, repeated): An unsigned packet number delta that indicates the number of contiguous packets being acked starting after the end of the previous gap. Along with the previous field, this field is repeated "Num Blocks" times.

- Timestamp Section:

  * Num Timestamps: An unsigned 8-bit number specifying the total number of <packet number, timestamp> pairs following, including the First Timestamp.

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

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

  * Delta Largest Observed (opt, repeated): (Same as above.)

  * Time Since Previous Timestamp (opt, repeated): An optional 16-bit unsigned value specifying time delta from the previous reported timestamp. It is encoded in the same format as the Ack Delay. Along with the previous field, this field is repeated "Num Timestamps" times.

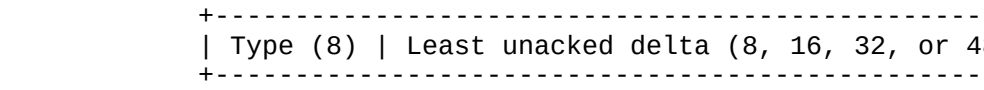
6.2.1. Time Format

DISCUSS_AND_REPLACE: Perhaps make this format simpler.

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

6.3. STOP_WAITING Frame

The STOP_WAITING frame is sent to inform the peer that it should not continue to wait for packets with packet numbers lower than a specified value. The packet number is encoded in 1, 2, 4 or 6 bytes, using the same coding length as is specified for the packet number for the enclosing packet's header (specified in the QUIC Frame packet's Flags field.) The frame is as follows:



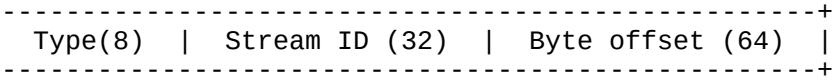
The fields in the STOP_WAITING frame are as follows:

- Frame Type: The Frame Type byte is an 8-bit value that must be set to 0x06 indicating that this is a STOP_WAITING frame.

- Least Unacked Delta: A variable-length packet number delta with the same length as the packet header's packet number. Subtract it from the complete packet number of the enclosing packet to determine the least unacked packet number. The resulting least unacked packet number is the earliest packet for which the sender is still awaiting an ack. If the receiver is missing any packets earlier than this packet, the receiver SHOULD consider those packets to be irrecoverably lost and MUST NOT report those packets as missing in subsequent acks.

6.4. WINDOW_UPDATE Frame

The WINDOW_UPDATE frame informs the peer of an increase in an endpoint's flow control receive window. The StreamID can be zero, indicating this WINDOW_UPDATE applies to the connection level flow control window, or non-zero, indicating that the specified stream should increase its flow control window. The frame is as follows:



The fields in the WINDOW_UPDATE frame are as follows:

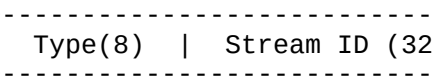
- Frame Type: The Frame Type byte is an 8-bit value that must be set to 0x04 indicating that this is a WINDOW_UPDATE frame.

- Stream ID: ID of the stream whose flow control windows is being updated, or 0 to specify the connection-level flow control window.

- Byte offset: A 64-bit unsigned integer indicating the absolute byte offset of data which can be sent on the given stream. In the case of connection level flow control, the cumulative number of bytes which can be sent on all currently open streams.

6.5. BLOCKED Frame

A sender sends a BLOCKED frame when it is ready to send data (and has data to send), but is currently flow control blocked. BLOCKED frames are purely informational frames, but extremely useful for debugging purposes. A receiver of a BLOCKED frame should simply discard it (after possibly printing a helpful log message). The frame is as follows:



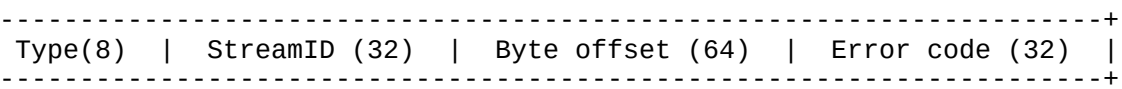
The fields in the BLOCKED frame are as follows:

- Frame Type: The Frame Type byte is an 8-bit value that must be set to 0x05 indicating that this is a BLOCKED frame.

- Stream ID: A 32-bit unsigned number indicating the stream which is flow control blocked. A non-zero Stream ID field specifies the stream that is flow control blocked. When zero, the Stream ID field indicates that the connection is flow control blocked.

6.6. RST_STREAM Frame

An endpoint may use a RST_STREAM frame to abruptly terminate a stream. The frame is as follows:

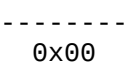


The fields are:

- Frame type: The Frame Type is an 8-bit value that must be set to 0x01 specifying that this is a RST_STREAM frame.
- Stream ID: The 32-bit Stream ID of the stream being terminated.
- Byte offset: A 64-bit unsigned integer indicating the absolute byte offset of the end of data written on this stream by the RST_STREAM sender.
- Error code: A 32-bit error code which indicates why the stream is being closed.

6.7. PADDING Frame

The PADDING frame pads a packet with 0x00 bytes. When this frame is encountered, the rest of the packet is expected to be padding bytes. The frame contains 0x00 bytes and extends to the end of the QUIC packet. A PADDING frame only has a Frame Type field, and must have the 8-bit Frame Type field set to 0x00.



6.8. PING frame

Endpoints can use PING frames to verify that their peers are still alive or to check reachability to the peer. The PING frame contains no payload. The receiver of a PING frame simply needs to ACK the packet containing this frame. The PING frame SHOULD be used to keep a connection alive when a stream is open. The default is to send a PING frame after 15 seconds of quiescence. A PING frame only has a Frame Type field, and must have the 8-bit Frame Type field set to 0x07.

```
   +--------+
   |  0x07  |
   +--------+
```

6.9. CONNECTION_CLOSE frame

An endpoint sends a CONNECTION_CLOSE frame to notify its peer that the connection is being closed. If there are open streams that haven't been explicitly closed, they are implicitly closed when the connection is closed. (Ideally, a GOAWAY frame would be sent with enough time that all streams are torn down.) The frame is as follows:

```
+-------------------------------------------------------------------------+
| Type(8) | Error code (32) | Reason phrase length (16) | Reason phrase |
+-------------------------------------------------------------------------+
```

The fields of a CONNECTION_CLOSE frame are as follows:

- Frame Type: An 8-bit value that must be set to 0x02 specifying that this is a CONNECTION_CLOSE frame.
- Error Code: A 32-bit error code which indicates the reason for closing this connection.
- Reason Phrase Length: A 16-bit unsigned number specifying the length of the reason phrase. This may be zero if the sender chooses to not give details beyond the QuicErrorCode.
- Reason Phrase: An optional human-readable explanation for why the connection was closed.

6.10. GOAWAY Frame

An endpoint may use a GOAWAY frame to notify its peer that the connection should stop being used, and will likely be aborted in the future. The endpoints will continue using any active streams, but the sender of the GOAWAY will not initiate any additional streams, and will not accept any new streams. The frame is as follows:

```
   +-----------------------------------------------------------+
   |  Type (8) |  Error code (32) |  Last Good Stream ID (32)  |
   +-----------------------------------------------------------+
   +-----------------------------------------------+
   | Reason phrase length (16)  |  Reason phrase  |
   +-----------------------------------------------+
```

The fields of a GOAWAY frame are as follows:

- Frame type: An 8-bit value that must be set to 0x03 specifying that this is a GOAWAY frame.

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

- Last Good Stream ID: The last Stream ID which was accepted by the sender of the GOAWAY message. If no streams were replied to, this value must be set to 0.

- Reason Phrase Length: A 16-bit unsigned number specifying the length of the reason phrase. This may be zero if the sender chooses to not give details beyond the error code.

- Reason Phrase: An optional human-readable explanation for why the connection was closed.

7. Packetization and Reliability

The maximum packet size for QUIC is the maximum size of the encrypted payload of the resulting UDP datagram. All QUIC packets SHOULD be sized to fit within the path's MTU to avoid IP fragmentation. The recommended default maximum packet size is 1350 bytes for IPv6 and 1370 bytes for IPv4. To optimize better, endpoints MAY use PLPMTUD [RFC4821] for detecting the path's MTU and setting the maximum packet size appropriately.

A sender bundles one or more frames in a Regular QUIC packet. A sender MAY bundle any set of frames in a packet. All QUIC packets MUST contain a packet number and MAY contain one or more frames (Section XX). Packet numbers MUST be unique within a connection and MUST NOT be reused within the same connection. Packet numbers MUST be assigned to packets in a strictly monotonically increasing order. The initial packet number used, at both the client and the server, MUST be 0. That is, the first packet in both directions of the connection MUST have a packet number of 0.

A sender SHOULD minimize per-packet bandwidth and computational costs by bundling as many frames as possible within a QUIC packet. A sender MAY wait for a short period of time to bundle multiple frames before sending a packet that is not maximally packed, to avoid sending out large numbers of small packets. An implementation may use heuristics about expected application sending behavior to determine whether and for how long to wait. This waiting period is an implementation decision, and an implementation should be careful to delay conservatively, since any delay is likely to increase application-visible latency.

Regular QUIC packets are "containers" of frames; a packet is never retransmitted whole, but frames in a lost packet may be rebundled and transmitted in a subsequent packet as necessary.

A packet may contain frames and/or application data, only some of which may require reliability. When a packet is detected as lost, the sender SHOULD only resend frames that require retransmission.

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

- ACK, STOP_WAITING, and PADDING frames MUST NOT be retransmitted. New frames of these types may however be bundled with any outgoing packet.

- All other frames MUST be retransmitted.

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

A receiver acknowledges receipt of a received packet by sending one or more ACK frames containing the packet number of the received packet. To avoid perpetual acking between endpoints, a receiver MUST NOT generate an ack in response to every packet containing only ACK frames. However, since it is possible that an endpoint sends only packets containing ACK frame (or other non-retransmittable frames), the receiving peer MAY send an ACK frame after a reasonable number (currently 20) of such packets have been received.

Strategies and implications of the frequency of generating acknowledgments are discussed in more detail in [draft-loss-recovery].

8. Streams: QUIC’s Data Structuring Abstraction

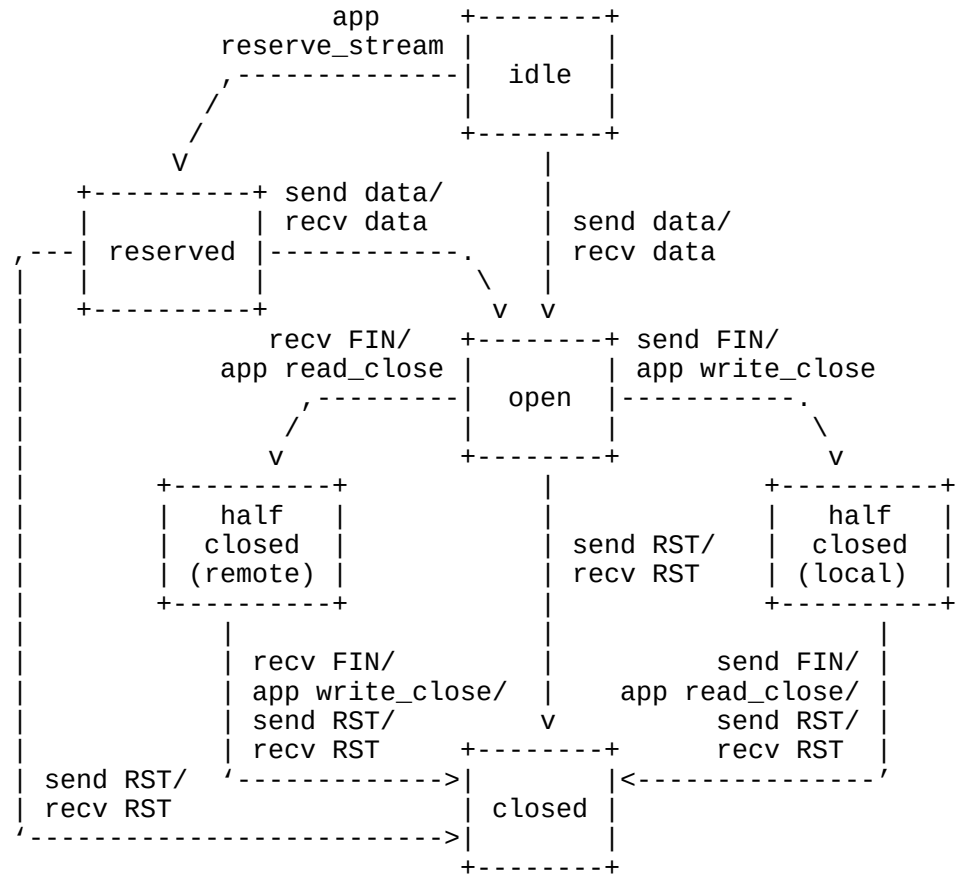
Streams in QUIC provide a lightweight, ordered, and bidirectional byte-stream abstraction. Streams can be created either by the client or the server, can concurrently send data interleaved with other streams, and can be cancelled. QUIC’s stream lifetime is modeled closely after HTTP/2’s [RFC7540]. Streams are independent of each other in delivery order. That is, data that is received on a stream is delivered in order within that stream, but there is no particular delivery order across streams. Transmit ordering among streams is left to the implementation. QUIC streams are considered lightweight

in that the creation and destruction of streams are expected to have minimal bandwidth and computational cost. A single STREAM frame may create, carry data for, and terminate a stream, or a stream may last the entire duration of a connection. Implementations are therefore advised to keep these extremes in mind and to implement stream creation and destruction to be as lightweight as possible.

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

8.1. Life of a Stream

The semantics of QUIC streams is based on HTTP/2 streams, and the lifecycle of a QUIC stream therefore closely follows that of an HTTP/2 stream [RFC7540], with some differences to accommodate the possibility of out-of-order delivery due to the use of multiple streams in QUIC. The lifecycle of a QUIC stream is shown in the following figure and described below.



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

data: application data in a STREAM frame
FIN: FIN flag in a STREAM frame
RST: RST_STREAM frame

app: application API signals to QUIC
reserve_stream: causes a StreamID to be reserved for later use
read_close: causes stream to be half-closed without receiving a FIN
write_close: causes stream to be half-closed without sending a FIN

Note that this diagram shows stream state transitions and the frames and flags that affect those transitions only. For the purpose of state transitions, the FIN flag is processed as a separate event to the frame that bears it; a STREAM frame with the FIN flag set can cause two state transitions. When the FIN bit is sent on an empty

STREAM frame, the offset in the STREAM frame MUST be one greater than the last data byte sent on this stream.

Both endpoints have a subjective view of the state of a stream that could be different when frames are in transit. Endpoints do not coordinate the creation of streams; they are created unilaterally by either endpoint. The negative consequences of a mismatch in states are limited to the "closed" state after sending RST_STREAM, where frames might be received for some time after closing.

Streams have the following states:

8.1.1. idle

All streams start in the "idle" state.

The following transitions are valid from this state:

Sending or receiving a STREAM frame causes the stream to become "open". The stream identifier is selected as described in Section XX. The same STREAM frame can also cause a stream to immediately become "half-closed".

An application can reserve an idle stream for later use. The stream state for the reserved stream transitions to "reserved".

Receiving any frame other than STREAM or RST_STREAM on a stream in this state MUST be treated as a connection error (Section XX) of type YYYY.

8.1.2. reserved

A stream in this state has been reserved for later use by the application. In this state only the following transitions are possible:

- Sending or receiving a STREAM frame causes the stream to become "open".
- Sending or receiving a RST_STREAM frame causes the stream to become "closed".

8.1.3. open

A stream in the "open" state may be used by both peers to send frames of any type. In this state, a sending peer must observe the flow-control limit advertised by its receiving peer (Section XX).

From this state, either endpoint can send a frame with the FIN flag set, which causes the stream to transition into one of the "half-closed" states. An endpoint sending an FIN flag causes the stream state to become "half-closed (local)". An endpoint receiving a FIN flag causes the stream state to become "half-closed (remote)"; the receiving endpoint MUST NOT process the FIN flag until all preceding data on the stream has been received.

Either endpoint can send a RST_STREAM frame from this state, causing it to transition immediately to "closed".

8.1.4. half-closed (local)

A stream that is in the "half-closed (local)" state MUST NOT be used for sending STREAM frames; WINDOW_UPDATE and RST_STREAM MAY be sent in this state.

A stream transitions from this state to "closed" when a frame that contains an FIN flag is received or when either peer sends a RST_STREAM frame.

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

8.1.5. half-closed (remote)

A stream that is "half-closed (remote)" is no longer being used by the peer to send any data. In this state, a sender is no longer obligated to maintain a receiver stream-level flow-control window.

If an endpoint receives any STREAM frames for a stream that is in this state, it MUST close the connection with a QUIC_STREAM_DATA_AFTER_TERMINATION error (Section XX).

A stream in this state can be used by the endpoint to send frames of any type. In this state, the endpoint continues to observe advertised stream-level and connection-level flow-control limits (Section XX).

A stream can transition from this state to "closed" by sending a frame that contains a FIN flag or when either peer sends a RST_STREAM frame.

8.1.6. closed

The "closed" state is the terminal state.

A final offset is present in both a frame bearing a FIN flag and in a RST_STREAM frame. Upon sending either of these frames for a stream, the endpoint MUST NOT send a STREAM frame carrying data beyond the final offset.

An endpoint that receives any frame for this stream after receiving either a FIN flag and all stream data preceding it, or a RST_STREAM frame, MUST quietly discard the frame, with one exception. If a STREAM frame carrying data beyond the received final offset is received, the endpoint MUST close the connection with a QUIC_STREAM_DATA_AFTER_TERMINATION error (Section XX).

An endpoint that receives a RST_STREAM frame (and which has not sent a FIN or a RST_STREAM) MUST immediately respond with a RST_STREAM frame, and MUST NOT send any more data on the stream. This endpoint may continue receiving frames for the stream on which a RST_STREAM is received.

If this state is reached as a result of sending a RST_STREAM frame, the peer that receives the RST_STREAM might have already sent -- or enqueued for sending -- frames on the stream that cannot be withdrawn. An endpoint MUST ignore frames that it receives on closed streams after it has sent a RST_STREAM frame. An endpoint MAY choose to limit the period over which it ignores frames and treat frames that arrive after this time as being in error.

STREAM frames received after sending RST_STREAM are counted toward the connection and stream flow-control windows. Even though these frames might be ignored, because they are sent before their sender receives the RST_STREAM, the sender will consider the frames to count against its flow-control windows.

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

(TODO: QUIC_STREAM_NO_ERROR is a special case. Write it up.)

8.2. Stream Identifiers

Streams are identified by an unsigned 32-bit integer, referred to as the StreamID. To avoid StreamID collision, clients MUST initiate

streams usinge odd-numbered StreamIDs; streams initiated by the server MUST use even-numbered StreamIDs.

A StreamID of zero (0x0) is reserved and used for connection-level flow control frames (Section XX); the StreamID of zero cannot be used to establish a new stream.

StreamID 1 (0x1) is reserved for the crypto handshake. StreamID 1 MUST NOT be used for application data, and MUST be the first client-initiated stream.

Streams MUST be created or reserved in sequential order, but MAY be used in arbitrary order. A QUIC endpoint MUST NOT reuse a StreamID on a given connection.

8.3. Stream Concurrency

An endpoint can limit the number of concurrently active incoming streams by setting the MSPC parameter (see Section XX) in the transport parameters. The maximum concurrent streams setting is specific to each endpoint and applies only to the peer that receives the setting. That is, clients specify the maximum number of concurrent streams the server can initiate, and servers specify the maximum number of concurrent streams the client can initiate.

Streams that are in the "open" state or in either of the "half-closed" states count toward the maximum number of streams that an endpoint is permitted to open. Streams in any of these three states count toward the limit advertised in the MSPC setting.

Endpoints MUST NOT exceed the limit set by their peer. An endpoint that receives a STREAM frame that causes its advertised concurrent stream limit to be exceeded MUST treat this as a stream error of type QUIC_TOO_MANY_OPEN_STREAMS (Section XX).

8.4. Sending and Receiving Data

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

When new data is to be sent on a stream, a sender MUST set the encapsulating STREAM frame’s offset field to the stream offset of the first byte of this new data. The first byte of data that is sent on

a stream has the stream offset 0. A receiver MUST ensure that received stream data is delivered to the application as an ordered byte-stream. Data received out of order MUST be buffered for later delivery, as long as it is not in violation of the receiver's flow control limits.

An endpoint MUST NOT send any stream data without consulting the congestion controller and the flow controller, with the following two exceptions.

- The crypto handshake stream, Stream 1, MUST NOT be subject to congestion control or connection-level flow control, but MUST be subject to stream-level flow control.

- An application MAY exclude specific stream IDs from connection-level flow control. If so, these streams MUST NOT be subject to connection-level flow control.

Flow control is described in detail in Section XX, and congestion control is described in the companion document [draft-iyengar-quic-loss-recovery].

9. Flow Control

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

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

A receiver sends WINDOW_UPDATE frames to the sender to advertise additional credit, for both connection and stream flow control. A receiver advertises the maximum absolute byte offset in the stream or in the connection which the receiver is willing to receive.

The initial flow control credit is 65536 bytes for both the stream and connection flow controllers.

A receiver MAY advertise a larger offset at any point in the connection by sending a WINDOW_UPDATE frame. A receiver MUST NOT renege on an advertisement; that is, once a receiver advertises an offset via a WINDOW_UPDATE frame, it MUST NOT subsequently advertise a smaller offset. A sender may receive WINDOW_UPDATE frames out of order; a sender MUST therefore ignore any reductions in flow control credit.

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

A receiver advertises credit for a stream by sending a WINDOW_UPDATE frame with the StreamID set appropriately. A receiver may simply use the current received offset to determine the flow control offset to be advertised.

Connection flow control is a limit to the total bytes of stream data sent in STREAM frames. A receiver advertises credit for a connection by sending a WINDOW_UPDATE frame with the StreamID set to zero (0x00). A receiver may maintain a cumulative sum of bytes received cumulatively on all streams to determine the value of the connection flow control offset to be advertised in WINDOW_UPDATE frames. A sender may maintain a cumulative sum of stream data bytes sent to impose the connection flow control limit.

9.1. Edge Cases and Other Considerations

There are some edge cases which must be considered when dealing with stream and connection level flow control. Given enough time, both endpoints must agree on flow control state. If one end believes it can send more than the other end is willing to receive, the connection will be torn down when too much data arrives. Conversely if a sender believes it is blocked, while endpoint B expects more data can be received, then the connection can be in a deadlock, with the sender waiting for a WINDOW_UPDATE which will never come.

9.1.1. Mid-stream RST_STREAM

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

were sent on the stream to make the same adjustment in its connection flow controller.

To avoid this de-synchronization, a RST_STREAM sender MUST include the final byte offset sent on the stream in the RST_STREAM frame. On receiving a RST_STREAM frame, a receiver definitively knows how many bytes were sent on that stream before the RST_STREAM frame, and the receiver MUST use the final offset to account for all bytes sent on the stream in its connection level flow controller.

9.1.2. Response to a RST_STREAM

Since streams are bidirectional, a sender of a RST_STREAM needs to know how many bytes the peer has sent on the stream. If an endpoint receives a RST_STREAM frame and has sent neither a FIN nor a RST_STREAM, it MUST send a RST_STREAM in response, bearing the offset of the last byte sent on this stream as the final offset.

9.1.3. Offset Increment

This document leaves when and how many bytes to advertise in a WINDOW_UPDATE to the implementation, but offers a few considerations. WINDOW_UPDATE frames constitute overhead, and therefore, sending a WINDOW_UPDATE with small offset increments is undesirable. At the same time, sending WINDOW_UPDATES with large offset increments requires the sender to commit to that amount of buffer. Implementations must find the correct tradeoff between these sides to determine how large an offset increment to send in a WINDOW_UPDATE.

A receiver MAY use an autotuning mechanism to tune the size of the offset increment to advertise based on a roundtrip time estimate and the rate at which the receiving application consumes data, similar to common TCP implementations.

9.1.4. BLOCKED frames

If a sender does not receive a WINDOW_UPDATE frame when it has run out of flow control credit, the sender will be blocked and MUST send a BLOCKED frame. A BLOCKED frame is expected to be useful for debugging at the receiver. A receiver SHOULD NOT wait for a BLOCKED frame before sending with a WINDOW_UPDATE, since doing so will cause at least one roundtrip of quiescence. For smooth operation of the congestion controller, it is generally considered best to not let the sender go into quiescence if avoidable. To avoid blocking a sender, and to reasonably account for the possibiity of loss, a receiver should send a WINDOW_UPDATE frame at least two roundtrips before it expects the sender to get blocked.

10. Error Codes

This section lists all the QUIC error codes that may be used in a CONNECTION_CLOSE frame. TODO: Trim list and group errors for readabiity.

- 0x01: QUIC_INTERNAL_ERROR. (Connection has reached an invalid state.)
- 0x02: QUIC_STREAM_DATA_AFTER_TERMINATION. (There were data frames after the a fin or reset.)
- 0x03: QUIC_INVALID_PACKET_HEADER. (Control frame is malformed.)
- 0x04: QUIC_INVALID_FRAME_DATA. (Frame data is malformed.)
- 0x30: QUIC_MISSING_PAYLOAD. (The packet contained no payload.)
- 0x2e: QUIC_INVALID_STREAM_DATA. (STREAM frame data is malformed.)
- 0x57: QUIC_OVERLAPPING_STREAM_DATA. (STREAM frame data overlaps with buffered data.)
- 0x3d: QUIC_UNENCRYPTED_STREAM_DATA. (Received STREAM frame data is not encrypted.)
- 0x58: QUIC_ATTEMPT_TO_SEND_UNENCRYPTED_STREAM_DATA. (Attempt to send unencrypted STREAM frame. Not sent on the wire, used for local logging.)
- 0x59: QUIC_MAYBE_CORRUPTED_MEMORY. (Received a frame which is likely the result of memory corruption.)
- 0x06: QUIC_INVALID_RST_STREAM_DATA. (RST_STREAM frame data is malformed.)
- 0x07: QUIC_INVALID_CONNECTION_CLOSE_DATA. (CONNECTION_CLOSE frame data is malformed.)
- 0x08: QUIC_INVALID_GOAWAY_DATA. (GOAWAY frame data is malformed.)
- 0x39: QUIC_INVALID_WINDOW_UPDATE_DATA. (WINDOW_UPDATE frame data is malformed.)
- 0x3a: QUIC_INVALID_BLOCKED_DATA. (BLOCKED frame data is malformed.)

- 0x3c: QUIC_INVALID_STOP_WAITING_DATA. (STOP_WAITING frame data is malformed.)
- 0x4e: QUIC_INVALID_PATH_CLOSE_DATA. (PATH_CLOSE frame data is malformed.)
- 0x09: QUIC_INVALID_ACK_DATA. (ACK frame data is malformed.)
- 0x0a: QUIC_INVALID_VERSION_NEGOTIATION_PACKET. (Version negotiation packet is malformed.)
- 0x0b: QUIC_INVALID_PUBLIC_RST_PACKET. (Public RST packet is malformed.)
- 0x0c: QUIC_DECRYPTION_FAILURE. (There was an error decrypting.)
- 0x0d: QUIC_ENCRYPTION_FAILURE. (There was an error encrypting.)
- 0x0e: QUIC_PACKET_TOO_LARGE. (The packet exceeded kMaxPacketSize.)
- 0x10: QUIC_PEER_GOING_AWAY. (The peer is going away. May be a client or server.)
- 0x11: QUIC_INVALID_STREAM_ID. (A stream ID was invalid.)
- 0x31: QUIC_INVALID_PRIORITY. (A priority was invalid.)
- 0x12: QUIC_TOO_MANY_OPEN_STREAMS. (Too many streams already open.)
- 0x4c: QUIC_TOO_MANY_AVAILABLE_STREAMS. (The peer created too many available streams.)
- 0x13: QUIC_PUBLIC_RESET. (Received public reset for this connection.)
- 0x14: QUIC_INVALID_VERSION. (Invalid protocol version.)
- 0x16: QUIC_INVALID_HEADER_ID. (The Header ID for a stream was too far from the previous.)
- 0x17: QUIC_INVALID_NEGOTIATED_VALUE. (Negotiable parameter received during handshake had invalid value.)
- 0x18: QUIC_DECOMPRESSION_FAILURE. (There was an error decompressing data.)

- 0x19: QUIC_NETWORK_IDLE_TIMEOUT. (The connection timed out due to no network activity.)
- 0x43: QUIC_HANDSHAKE_TIMEOUT. (The connection timed out waiting for the handshake to complete.)
- 0x1a: QUIC_ERROR_MIGRATING_ADDRESS. (There was an error encountered migrating addresses.)
- 0x56: QUIC_ERROR_MIGRATING_PORT. (There was an error encountered migrating port only.)
- 0x1b: QUIC_PACKET_WRITE_ERROR. (There was an error while writing to the socket.)
- 0x33: QUIC_PACKET_READ_ERROR. (There was an error while reading from the socket.)
- 0x32: QUIC_EMPTY_STREAM_FRAME_NO_FIN. (We received a STREAM_FRAME with no data and no fin flag set.)
- 0x38: QUIC_INVALID_HEADERS_STREAM_DATA. (We received invalid data on the headers stream.)
- 0x3b: QUIC_FLOW_CONTROL_RECEIVED_TOO_MUCH_DATA. (The peer received too much data, violating flow control.)
- 0x3f: QUIC_FLOW_CONTROL_SENT_TOO_MUCH_DATA. (The peer sent too much data, violating flow control.)
- 0x40: QUIC_FLOW_CONTROL_INVALID_WINDOW. (The peer received an invalid flow control window.)
- 0x3e: QUIC_CONNECTION_IP_POOLED. (The connection has been IP pooled into an existing connection.)
- 0x44: QUIC_TOO_MANY_OUTSTANDING_SENT_PACKETS. (The connection has too many outstanding sent packets.)
- 0x45: QUIC_TOO_MANY_OUTSTANDING_RECEIVED_PACKETS. (The connection has too many outstanding received packets.)
- 0x46: QUIC_CONNECTION_CANCELLED. (The quic connection has been cancelled.)
- 0x47: QUIC_BAD_PACKET_LOSS_RATE. (Disabled QUIC because of high packet loss rate.)

- 0x49: QUIC_PUBLIC_RESETS_POST_HANDSHAKE. (Disabled QUIC because of too many PUBLIC_RESETs post handshake.)

- 0x4a: QUIC_TIMEOUTS_WITH_OPEN_STREAMS. (Disabled QUIC because of too many timeouts with streams open.)

- 0x4b: QUIC_FAILED_TO_SERIALIZE_PACKET. (Closed because we failed to serialize a packet.)

- 0x55: QUIC_TOO_MANY_RTOS. (QUIC timed out after too many RTOs.)

- 0x1c: QUIC_HANDSHAKE_FAILED. (Crypto errors.Hanshake failed.)

- 0x1d: QUIC_CRYPTO_TAGS_OUT_OF_ORDER. (Handshake message contained out of order tags.)

- 0x1e: QUIC_CRYPTO_TOO_MANY_ENTRIES. (Handshake message contained too many entries.)

- 0x1f: QUIC_CRYPTO_INVALID_VALUE_LENGTH. (Handshake message contained an invalid value length.)

- 0x20: QUIC_CRYPTO_MESSAGE_AFTER_HANDSHAKE_COMPLETE. (A crypto message was received after the handshake was complete.)

- 0x21: QUIC_INVALID_CRYPTO_MESSAGE_TYPE. (A crypto message was received with an illegal message tag.)

- 0x22: QUIC_INVALID_CRYPTO_MESSAGE_PARAMETER. (A crypto message was received with an illegal parameter.)

- 0x34: QUIC_INVALID_CHANNEL_ID_SIGNATURE. (An invalid channel id signature was supplied.)

- 0x23: QUIC_CRYPTO_MESSAGE_PARAMETER_NOT_FOUND. (A crypto message was received with a mandatory parameter missing.)

- 0x24: QUIC_CRYPTO_MESSAGE_PARAMETER_NO_OVERLAP. (A crypto message was received with a parameter that has no overlapwith the local parameter.)

- 0x25: QUIC_CRYPTO_MESSAGE_INDEX_NOT_FOUND. (A crypto message was received that contained a parameter with too fewvalues.)

- 0x5e: QUIC_UNSUPPORTED_PROOF_DEMAND. (A demand for an unsupport proof type was received.)

- 0x26: QUIC_CRYPTO_INTERNAL_ERROR. (An internal error occured in crypto processing.)
- 0x27: QUIC_CRYPTO_VERSION_NOT_SUPPORTED. (A crypto handshake message specified an unsupported version.)
- 0x48: QUIC_CRYPTO_HANDSHAKE_STATELESS_REJECT. (A crypto handshake message resulted in a stateless reject.)
- 0x28: QUIC_CRYPTO_NO_SUPPORT. (There was no intersection between the crypto primitives supported by thepeer and ourselves.)
- 0x29: QUIC_CRYPTO_TOO_MANY_REJECTS. (The server rejected our client hello messages too many times.)
- 0x2a: QUIC_PROOF_INVALID. (The client rejected the server’s certificate chain or signature.)
- 0x2b: QUIC_CRYPTO_DUPLICATE_TAG. (A crypto message was received with a duplicate tag.)
- 0x2c: QUIC_CRYPTO_ENCRYPTION_LEVEL_INCORRECT. (A crypto message was received with the wrong encryption level (i.e. itshould have been encrypted but was not.))
- 0x2d: QUIC_CRYPTO_SERVER_CONFIG_EXPIRED. (The server config for a server has expired.)
- 0x35: QUIC_CRYPTO_SYMMETRIC_KEY_SETUP_FAILED. (We failed to setup the symmetric keys for a connection.)
- 0x36: QUIC_CRYPTO_MESSAGE_WHILE_VALIDATING_CLIENT_HELLO. (A handshake message arrived, but we are still validating theprevious handshake message.)
- 0x41: QUIC_CRYPTO_UPDATE_BEFORE_HANDSHAKE_COMPLETE. (A server config update arrived before the handshake is complete.)
- 0x5a: QUIC_CRYPTO_CHLO_TOO_LARGE. (CHLO cannot fit in one packet.)
- 0x37: QUIC_VERSION_NEGOTIATION_MISMATCH. (This connection involved a version negotiation which appears to have beentampered with.)
- 0x50: QUIC_IP_ADDRESS_CHANGED. (IP address changed causing connection close.)

- 0x51: QUIC_CONNECTION_MIGRATION_NO_MIGRATABLE_STREAMS. (Connection migration errors.Network changed, but connection had no migratable streams.)

- 0x52: QUIC_CONNECTION_MIGRATION_TOO_MANY_CHANGES. (Connection changed networks too many times.)

- 0x53: QUIC_CONNECTION_MIGRATION_NO_NEW_NETWORK. (Connection migration was attempted, but there was no new network tomigrate to.)

- 0x54: QUIC_CONNECTION_MIGRATION_NON_MIGRATABLE_STREAM. (Network changed, but connection had one or more non-migratable streams.)

- 0x5d: QUIC_TOO_MANY_FRAME_GAPS. (Stream frames arrived too discontiguously so that stream sequencer buffermaintains too many gaps.)

- 0x5f: QUIC_STREAM_SEQUENCER_INVALID_STATE. (Sequencer buffer get into weird state where continuing read/write will leadto crash.)

- 0x60: QUIC_TOO_MANY_SESSIONS_ON_SERVER. (Connection closed because of server hits max number of sessions allowed.

11. Security and Privacy Considerations

11.1. Spoofed Ack Attack

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

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

The second mitigation is that the server can require that acks for sent packets match the encryption level of the sent packet. This mitigation is useful if the connection has an ephemeral forward-secure key that is generated and used for every new connection. If a packet sent is encrypted with a forward-secure key, then any acks

that are received for them must also be forward-secure encrypted. Since the attacker will not have the forward secure key, the attacker will not be able to generate forward-secure encrypted ack packets.

12. Contributors

This protocol is the outcome of work by many engineers, not just the authors of this document. The design and rationale behind QUIC draw significantly from work by Jim Roskind [1]. In alphabetical order, the contributors to the project are: Britt Cyr, Jeremy Dorfman, Ryan Hamilton, Jana Iyengar, Fedor Kouranov, Charles Krasic, Jo Kulik, Adam Langley, Jim Roskind, Robbie Shade, Satyam Shekhar, Cherie Shi, Ian Swett, Raman Tenneti, Victor Vasiliev, Antonio Vicente, Patrik Westin, Alyssa Wilk, Dale Worley, Fan Yang, Dan Zhang, Daniel Ziegler.

13. Acknowledgments

Special thanks are due to the following for helping shape QUIC and its deployment: Chris Bentzel, Misha Efimov, Roberto Peon, Alistair Riddoch, Siddharth Vijayakrishnan, and Assar Westerlund. QUIC has also benefited immensely from discussions with folks in private conversations and public ones on the proto-quic@chromium.org mailing list.

.

14. References

14.1. Normative References

[RFC2119] Bradner, S., "Key Words for use in RFCs to Indicate Requirement Levels", March 1997.

[draft-thomson-quic-tls] Thomson, M. and R. Hamilton, "Porting QUIC to TLS", March 2016.

[draft-iyengar-quic-loss-recovery] Iyengar, J. and I. Swett, "QUIC Loss Recovery and Congestion Control", July 2016.

14.2. Informative References

[RFC7540] Belshe, M., Peon, R., and M. Thomson, "Hypertext Transfer Protocol Version 2 (HTTP/2)", May 2015.

[QUIC-CRYPTO]
Langley, A. and W. Chang, "QUIC Crypto", June 2015, <http://goo.gl/OuVSxa>.

14.3. URIs

[1] https://goo.gl/dMVtFi

Authors' Addresses

Ryan Hamilton
Google

Email: rch@google.com

Janardhan Iyengar
Google

Email: jri@google.com

Ian Swett
Google

Email: ianswett@google.com

Alyssa Wilk
Google

Email: alyssar@google.com

Network Working Group
Internet-Draft
Intended status: Experimental
Expires: May 4, 2017

J. Iyengar
I. Swett
Google

October 31, 2016

QUIC Congestion Control And Loss Recovery
draft-iyengar-quic-loss-recovery-01

Abstract

QUIC is a new multiplexed and secure transport atop UDP. QUIC builds on decades of transport and security experience, and implements mechanisms that make it attractive as a modern general-purpose transport. QUIC implements the spirit of known TCP loss detection mechanisms, described in RFCs, various Internet-drafts, and also those prevalent in the Linux TCP implementation. This document describes QUIC loss detection and congestion control, and attributes the TCP equivalent in RFCs, Internet-drafts, academic papers, and TCP implementations.

Status of This Memo

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

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

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

This Internet-Draft will expire on May 4, 2017.

1. Introduction

QUIC is a new multiplexed and secure transport atop UDP. QUIC builds on decades of transport and security experience, and implements mechanisms that make it attractive as a modern general-purpose transport. The QUIC protocol is described in [draft-hamilton-quic-transport-protocol].

QUIC implements the spirit of known TCP loss recovery mechanisms, described in RFCs, various Internet-drafts, and also those prevalent in the Linux TCP implementation. This document describes QUIC congestion control and loss recovery, and where applicable, attributes the TCP equivalent in RFCs, Internet-drafts, academic papers, and/or TCP implementations.

This document first describes pre-requisite parts of the QUIC transmission machinery, then discusses QUIC's default congestion control and loss detection mechanisms, and finally lists the various TCP mechanisms that QUIC loss detection implements (in spirit.)

2. Design of the QUIC Transmission Machinery

All transmissions in QUIC are sent with a packet-level header, which includes a packet sequence number (referred to below as a packet number). These packet numbers never repeat in the lifetime of a connection, and are monotonically increasing, which makes duplicate detection trivial. This fundamental design decision obviates the need for disambiguating between transmissions and retransmissions and eliminates significant complexity from QUIC's interpretation of TCP loss detection mechanisms.

Every packet may contain several frames. We outline the frames that are important to the loss detection and congestion control machinery below.

- Retransmittable frames are frames requiring reliable delivery. The most common are STREAM frames, which typically contain application data.

- Crypto handshake data is also sent as STREAM data, and uses the reliability machinery of QUIC underneath.

- ACK frames contain acknowledgment information. QUIC uses a SACK-based scheme, where acks express up to 256 ranges. The ACK frame also includes a receive timestamp for each packet newly acked.

2.1. Relevant Differences Between QUIC and TCP

There are some notable differences between QUIC and TCP which are important for reasoning about the differences between the loss recovery mechanisms employed by the two protocols. We briefly describe these differences below.

2.1.1. Monotonically Increasing Packet Numbers

TCP conflates transmission sequence number at the sender with delivery sequence number at the receiver, which results in retransmissions of the same data carrying the same sequence number, and consequently to problems caused by "retransmission ambiguity". QUIC separates the two: QUIC uses a packet sequence number (referred to as the "packet number") for transmissions, and any data that is to be delivered to the receiving application(s) is sent in one or more streams, with stream offsets encoded within STREAM frames inside of packets that determine delivery order.

QUIC’s packet number is strictly increasing, and directly encodes transmission order. A higher QUIC packet number signifies that the packet was sent later, and a lower QUIC packet number signifies that the packet was sent earlier. When a packet containing frames is deemed lost, QUIC rebundles necessary frames in a new packet with a new packet number, removing ambiguity about which packet is acknowledged when an ACK is received. Consequently, more accurate RTT measurements can be made, spurious retransmissions are trivially detected, and mechanisms such as Fast Retransmit can be applied universally, based only on packet number.

This design point significantly simplifies loss detection mechanisms for QUIC. Most TCP mechanisms implicitly attempt to infer transmission ordering based on TCP sequence numbers --- a non-trivial task, especially when TCP timestamps are not available.

2.1.2. No Reneging

QUIC ACKs contain information that is equivalent to TCP SACK, but QUIC does not allow any acked packet to be reneged, greatly simplifying implementations on both sides and reducing memory pressure on the sender.

2.1.3. More ACK Ranges

QUIC supports up to 256 ACK ranges, opposed to TCP's 3 SACK ranges. In high loss environments, this speeds recovery.

2.1.4. Explicit Correction For Delayed Acks

QUIC ACKs explicitly encode the delay incurred at the receiver between when a packet is received and when the corresponding ACK is sent. This allows the receiver of the ACK to adjust for receiver delays, specifically the delayed ack timer, when estimating the path RTT. This mechanism also allows a receiver to measure and report the delay from when a packet was received by the OS kernel, which is useful in receivers which may incur delays such as context-switch latency before a userspace QUIC receiver processes a received packet.

3. Loss Detection

We now describe QUIC's loss detection as functions that should be called on packet transmission, when a packet is acked, and timer expiration events.

3.1. Variables of interest

We first describe the variables required to implement the loss detection mechanisms described in this section.

- loss_detection_alarm: Multi-modal alarm used for loss detection.
- alarm_mode: QUIC maintains a single loss detection alarm, which switches between various modes. This mode is used to determine the duration of the alarm.
- handshake_count: The number of times the handshake packets have been retransmitted without receiving an ack.
- tlp_count: The number of times a tail loss probe has been sent without receiving an ack.
- rto_count: The number of times an rto has been sent without receiving and ack.
- smoothed_rtt: The smoothed RTT of the connection, computed as described in [RFC 6298]. TODO: Describe RTT computations.
- reordering_threshold: The largest delta between the largest acked retransmittable packet and a packet containing retransmittable frames before it's declared lost.

- time_loss: When true, loss detection operates solely based on reordering threshold in time, rather than in packet number gaps.

3.2. Initialization

At the beginning of the connection, initialize the loss detection variables as follows:

```
loss_detection_alarm.reset();
handshake_count = 0;
tlp_count = 0;
rto_count = 0;
smoothed_rtt = 0;
reordering_threshold = 3;
time_loss = false;
```

3.3. Setting the Loss Detection Alarm

QUIC loss detection uses a single alarm for all timer-based loss detection. The duration of the alarm is based on the alarm's mode, which is set in the packet and timer events further below. The function SetLossDetectionAlarm defined below shows how the single timer is set based on the alarm mode.

Pseudocode for SetLossDetectionAlarm follows.

```
  SetLossDetectionAlarm():
    if (retransmittable packets are outstanding):

      loss_detection_alarm.cancel();
      return;
    if (handshake packets are outstanding):

      alarm_duration = max(1.5 * smoothed_rtt, 10ms) << handshake_count;
      handshake_count++;
    else if (largest sent packet is acked):
      // Set alarm based on short timer for early retransmit.
      alarm_duration = 0.25 x smoothed_rtt;
    else if (tlp_count < 2):

      if (retransmittable_packets_outstanding = 1):
        alarm_duration = max(1.5 x smoothed_rtt + delayed_ack_timer,
                             2 x smoothed_rtt);
      else:
        alarm_duration = max (10ms, 2 x smoothed_rtt);
      tlp_count++;
    else:

      if (rto_count = 0):
        alarm_duration = max(200ms, smoothed_rtt + 4 x rttvar);
      else:
        alarm_duration = loss_detection_alarm.get_delay() << 1;

      rto_count++;


    loss_detecton_alarm.set(now + alarm_duration);
```

3.4. On Sending a Packet

After any packet is sent, be it a new transmission or a rebundled transmission, the following OnPacketSent function is called. The parameters to OnPacketSent are as follows.

- packet_number: The packet number of the sent packet.

- is_retransmittble: A boolean that indicates whether the packet contains at least one frame requiring reliable deliver. The retransmittability of various QUIC frames is described in [draft-hamilton-quic-protocol]. If false, it is still acceptable for an ack to be received for this packet. However, a caller MUST NOT set is_retransmittable to true if an ack is not expected.

Pseudocode for OnPacketSent follows.

```
OnPacketSent(packet_number, is_retransmittable):
  if is_retransmittable:
    SetLossDetectionAlarm()
```

3.5. On Packet Acknowledgment

When a packet is acked for the first time, the following OnPacketAcked function is called. Note that a single ACK frame may newly acknowledge several packets. OnPacketAcked must be called once for each of these newly acked packets.

OnPacketAcked takes one parameter, acked_packet, which is the packet number of the newly acked packet, and returns a list of packet numbers that are detected as lost.

Pseudocode for OnPacketAcked follows.

```
OnPacketAcked(acked_packet):
  handshake_count = 0;
  tlp_count = 0;
  rto_count = 0;
  UpdateRtt(); // TODO: document RTT estimator.
  DetectLostPackets(acked_packet);
  SetLossDetectionAlarm();
```

3.6. On Alarm Firing

QUIC uses one loss recovery alarm, which when set, can be in one of several modes. When the alarm fires, the mode determines the action to be performed. OnAlarm returns a list of packet numbers that are detected as lost.

Pseudocode for OnAlarm follows.

```
OnAlarm(acked_packet):
  lost_packets = DetectLostPackets(acked_packet);
  MaybeRetransmitLostPackets();
  SetLossDetectionAlarm();
```

3.7. Detecting Lost Packets

Packets in QUIC are only considered lost once a larger packet number is acknowledged. DetectLostPackets is called every time there is a new largest packet or if the loss detection alarm fires the previous largest acked packet is supplied.

DetectLostPackets takes one parameter, acked_packet, which is the packet number of the largest acked packet, and returns a list of packet numbers detected as lost.

Pseudocode for DetectLostPackets follows.

```
DetectLostPackets(acked_packet):
  lost_packets = {};
  foreach (unacked_packet less than acked_packet):
      if (unacked_packet.time_sent <
          acked_packet.time_sent - 1/8 * smoothed_rtt):
        lost_packets.insert(unacked_packet.packet_number);
    else if (unacked_packet.packet_number <
             acked_packet.packet_number - reordering_threshold)
      lost_packets.insert(unacked_packet.packet_number);
  return lost_packets;
```

4. Congestion Control

(describe NewReno-style congestion control for QUIC.)

5. TCP mechanisms in QUIC

QUIC implements the spirit of a variety of RFCs, Internet drafts, and other well-known TCP loss recovery mechanisms, though the implementation details differ from the TCP implementations.

5.1. RFC 6298 (RTO computation)

QUIC calculates SRTT and RTTVAR according to the standard formulas. An RTT sample is only taken if the delayed ack correction is smaller than the measured RTT (otherwise a negative RTT would result), and the ack's contains a new, larger largest observed packet number. min_rtt is only based on the observed RTT, but SRTT uses the delayed ack correction delta.

As described above, QUIC implements RTO with the standard timeout and CWND reduction. However, QUIC retransmits the earliest outstanding packets rather than the latest, because QUIC doesn't have retransmission ambiguity. QUIC uses the commonly accepted min RTO of 200ms instead of the 1s the RFC specifies.

5.2. FACK Loss Recovery (paper)

QUIC implements the algorithm for early loss recovery described in the FACK paper (and implemented in the Linux kernel.) QUIC uses the packet number to measure the FACK reordering threshold. Currently

QUIC does not implement an adaptive threshold as many TCP implementations(ie: the Linux kernel) do.

5.3. RFC 3782, RFC 6582 (NewReno Fast Recovery)

QUIC only reduces its CWND once per congestion window, in keeping with the NewReno RFC. It tracks the largest outstanding packet at the time the loss is declared and any losses which occur before that packet number are considered part of the same loss event. It's worth noting that some TCP implementations may do this on a sequence number basis, and hence consider multiple losses of the same packet a single loss event.

5.4. TLP (draft)

QUIC always sends two tail loss probes before RTO is triggered. QUIC invokes tail loss probe even when a loss is outstanding, which is different than some TCP implementations.

5.5. RFC 5827 (Early Retransmit) with Delay Timer

QUIC implements early retransmit with a timer in order to minimize spurious retransmits. The timer is set to 1/4 SRTT after the final outstanding packet is acked.

5.6. RFC 5827 (F-RTO)

QUIC implements F-RTO by not reducing the CWND and SSThresh until a subsequent ack is received and it's sure the RTO was not spurious. Conceptually this is similar, but it makes for a much cleaner implementation with fewer edge cases.

5.7. RFC 6937 (Proportional Rate Reduction)

PRR-SSRB is implemented by QUIC in the epoch when recovering from a loss.

5.8. TCP Cubic (draft) with optional RFC 5681 (Reno)

TCP Cubic is the default congestion control algorithm in QUIC. Reno is also an easily available option which may be requested via connection options and is fully implemented.

5.9. Hybrid Slow Start (paper)

QUIC implements hybrid slow start, but disables ack train detection, because it has shown to falsely trigger when coupled with packet pacing, which is also on by default in QUIC. Currently the minimum

delay increase is 4ms, the maximum is 16ms, and within that range QUIC exits slow start if the min_rtt within a round increases by more than ⅛ of the connection mi

5.10. RACK (draft)

QUIC's loss detection is by it's time-ordered nature, very similar to RACK. Though QUIC defaults to loss detection based on reordering threshold in packets, it could just as easily be based on fractions of an rtt, as RACK does.

n

_rtt.

6. References

6.1. Normative References

[RFC2119] Bradner, S., "Key Words for use in RFCs to Indicate Requirement Levels", March 1997.

6.2. Informative References

[draft-hamilton-quic-transport-protocol] Hamilton, R., Iyengar, J., Swett, I., and A. Wilk, "QUIC: A UDP-Based Multiplexed and Secure Transport", July 2016.

Authors' Addresses

Janardhan Iyengar
Google

Email: jri@google.com

Ian Swett
Google

Email: ianswett@google.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: January 9, 2017

R. Shade
M. Warres
Google

July 8, 2016

HTTP/2 Semantics Using The QUIC Transport Protocol
draft-shade-quic-http2-mapping-00

Abstract

The QUIC transport protocol has several features that are desirable in a transport for HTTP/2, such as stream multiplexing, per-stream flow control, and low-latency connection establishment. This document describes a mapping of HTTP/2 semantics over QUIC. Specifically, this document identifies HTTP/2 features that are subsumed by QUIC, and describes how the other features can be implemented atop QUIC.

Status of This Memo

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

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

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

This Internet-Draft will expire on January 9, 2017.

Copyright Notice

Table of Contents

1. Introduction

The QUIC transport protocol has several features that are desirable in a transport for HTTP/2, such as stream multiplexing, per-stream flow control, and low-latency connection establishment. This document describes a mapping of HTTP/2 semantics over QUIC. Specifically, this document identifies HTTP/2 features that are subsumed by QUIC, and describes how the other features can be implemented atop QUIC.

QUIC is described in [draft-hamilton-quic-transport-protocol]. For a full description of HTTP/2, see [RFC 7540].

2. QUIC advertisement

A server advertises that it can speak HTTP/2-over-QUIC via the Alt-Svc HTTP response header. It does so by including the header in any response sent over a non-QUIC (e.g. HTTP/2 over TLS) connection:

Alt-Svc: quic=":443"

In addition, the list of QUIC versions supported by the server can be specified by the v= parameter. For example, if a server supported both version 33 and 34 it would specify the following header:

```
Alt-Svc: quic=":443"; v="34,33"
```

On receipt of this header, a client may attempt to establish a QUIC connection on port 443 and, if successful, send HTTP/2 requests using the mapping described in this document.

Connectivity problems (e.g. firewall blocking UDP) may result in QUIC connection establishment failure, in which case the client should gracefully fallback to HTTP/2-over-TLS/TCP.

3. Connection establishment

HTTP/2-over-QUIC connections are established as described in [draft-hamilton-quic-transport-protocol]. The QUIC crypto handshake MUST use TLS [draft-thomson-quic-tls].

While connection-level options pertaining to the core QUIC protocol are set in the initial crypto handshake [Combined Crypto and Transport Handshake], HTTP/2-specific settings are conveyed in the HTTP/2 SETTINGS frame. After the QUIC connection is established, an HTTP/2 SETTINGS frame may be sent as the initial frame of the QUIC headers stream (StreamID 3, See [Stream Mapping]). As in HTTP/2, additional SETTINGS frames may be sent mid-connection by either endpoint.

TODO: decide whether to acknowledge receipt of SETTINGS through empty SETTINGS frames with ACK bit set, as in HTTP/2, or rely on transport-level acknowledgment.

Some transport-level options that HTTP/2-over-TCP specifies via the SETTINGS frame are superseded by QUIC transport parameters in HTTP/2-over-QUIC. Below is a listing of how each HTTP/2 SETTINGS parameter is mapped:

- SETTINGS_HEADER_TABLE_SIZE
  - Sent in HTTP/2 SETTINGS frame.
- SETTINGS_ENABLE_PUSH
  - Sent in HTTP/2 SETTINGS frame [TBD, currently set using QUIC "SPSH" connection option]
- SETTINGS_MAX_CONCURRENT_STREAMS

    * QUIC requires the maximum number of incoming streams per connection to be specified in the initial crypto handshake, using the "MSPC" tag. Specifying SETTINGS_MAX_CONCURRENT_STREAMS in the HTTP/2 SETTINGS frame is an error.

- SETTINGS_INITIAL_WINDOW_SIZE

    * QUIC requires both stream and connection flow control window sizes to be specified in the initial crypto handshake, using the "SFCW" and "CFCW" tags, respectively. Specifying SETTINGS_INITIAL_WINDOW_SIZE in the HTTP/2 SETTINGS frame is an error.

- SETTINGS_MAX_FRAME_SIZE

    * This setting has no equivalent in QUIC. Specifying it in the HTTP/2 SETTINGS frame is an error.

- SETTINGS_MAX_HEADER_LIST_SIZE

    * Sent in HTTP/2 SETTINGS frame.

As with HTTP/2-over-TCP, unknown SETTINGS parameters are tolerated but ignored. SETTINGS parameters are acknowledged by the receiving peer, by sending an empty SETTINGS frame in response with the ACK bit set.

4. Sending a request on an HTTP/2-over-QUIC connection

A high level overview of sending an HTTP/2 request on an established QUIC connection is as follows, with further details in later sections of this document. A client should first encode any HTTP headers using HPACK [RFC7541] and frame them as HTTP/2 HEADERS frames. These are sent on StreamID 3 (see [Stream Mapping]). The exact layout of the HEADERS frame is described in Section 6.2 of [RFC7540]. No HTTP/2 padding is required: QUIC provides a PADDING frame for this purpose.

While HEADERS are sent on stream 3, the mandatory stream identifier in each HEADERS frame indicates the QUIC StreamID on which a corresponding request body may be sent. If there is no non-header data, the specified QUIC data stream will never be used.

4.1. Terminating a stream

A stream can be terminated in one of three ways:

- the request/response is headers only, in which case a HEADERS frame with the END_STREAM bit set ends the stream specified in the HEADERS frame
- the request/response has headers and body but no trailing headers, in which case the final QUIC STREAM frame will have the FIN bit set
- the request/response has headers, body, and trailing headers, in which case the final QUIC STREAM frame will not have the FIN bit set, and the trailing HEADERS frame will have the END_STREAM bit set

(TODO: Describe mapping of HTTP/2 stream state machine to QUIC stream state machine.)

5. Writing data to QUIC streams

A QUIC stream provides reliable in-order delivery of bytes, within that stream. On the wire, data is framed into QUIC STREAM frames, but this framing is invisible to the HTTP/2 layer. A QUIC receiver buffers and orders received STREAM frames, exposing the data contained within as a reliable byte stream to the application.

Bytes written to Stream 3 must be HTTP/2 HEADERS frames (or other HTTP/2 non-data frames), whereas bytes written to data streams should simply be request or response bodies. No further framing is required by HTTP/2 (i.e. no HTTP/2 DATA frames are used).

If data arrives on a data stream before the corresponding HEADERS have arrived on stream 3, then the data is buffered until the HEADERS arrive.

6. Stream Mapping

When HTTP/2 headers and data are sent over QUIC, the QUIC layer handles most of the stream management. HTTP/2 StreamIDs are replaced by QUIC StreamIDs. HTTP/2 does not need to do any explicit stream framing when using QUIC---data sent over a QUIC stream simply consists of HTTP/2 headers or body. Requests and responses are considered complete when the QUIC stream is closed in the corresponding direction.

Like HTTP/2, QUIC uses odd-numbered StreamIDs for client initiated streams, and even-numbered IDs for server initiated (i.e. server push) streams. Unlike HTTP/2 there are a couple of reserved (or dedicated) StreamIDs in QUIC.

6.1. Reserved Streams

StreamID 1 is reserved for crypto operations (the handshake, crypto config updates), and MUST NOT be used for HTTP/2 headers or body, see [core protocol doc]. StreamID 3 is reserved for sending and receiving HTTP/2 HEADERS frames. Therefore the first client initiated data stream has StreamID 5.

There are no reserved server initiated StreamIDs, so the first server initiated (i.e. server push) stream has an ID of 2, followed by 4, etc.

6.1.1. Stream 3: headers

HTTP/2-over-QUIC uses HPACK header compression as described in [RFC7541]. HPACK was designed for HTTP/2 with the assumption of in-order delivery such as that provided by TCP. A sequence of encoded header blocks must arrive (and be decoded) at an endpoint in the same order in which they were encoded. This ensures that the dynamic state at the two endpoints remains in sync.

QUIC streams provide in-order delivery of data sent on those streams, but there are no guarantees about order of delivery between streams. To achieve in-order delivery of HEADERS frames in QUIC, they are all sent on the reserved Stream 3. Data (request/response bodies) which arrive on other data streams are buffered until the corresponding HEADERS arrive and are read out of Stream 3.

This does introduce head-of-line blocking: if the packet containing HEADERS for stream N is lost or reordered then stream N+2 cannot be processed until they it has been retransmitted successfully, even though the HEADERS for stream N+2 may have arrived.

Trailing headers (trailers) can also be sent on stream 3. These are sent as HTTP/2 HEADERS frames, but MUST have the END_STREAM bit set, and MUST include a ":final-offset" pseudo-header. Since QUIC supports out of order delivery, receipt of a HEADERS frame with the END_STREAM bit set does not guarantee that the entire request/response body has been fully received. Therefore, the extra ":final-offset" pseudo-header is included in trailing HEADERS frames to indicate the total number of body bytes sent on the corresponding data stream. This is used by the QUIC layer to determine when the full request has been received and therefore when it is safe to tear

down local stream state. The ":final-offset" pseudo header is stripped from the HEADERS before passing to the HTTP/2 layer.

6.1.2. Stream states

The mapping of HTTP/2-over-QUIC with potential out of order delivery of HEADERS frames results in some changes to the HTTP/2 stream state transition diagram [https://tools.ietf.org/html/rfc7540#section-5.1]. Specifically the transition from "open" to "half closed (remote)", and the transition from "half closed (local)" to "closed" takes place only when:

- the peer has explicitly ended the stream via either
  - an HTTP/2 HEADERS frame with END_STREAM bit set and, in the case of trailing headers, the :final-offset pseudo-header
  - or a QUIC stream frame with the FIN bit set.
- and the full request or response body has been received.

7. Stream Priorities

HTTP/2-over-QUIC uses the HTTP/2 priority scheme described in [RFC7540 Section 5.3]. In the HTTP/2 priority scheme, a given stream can be designated as dependent upon another stream, which expresses the preference that the latter stream (the "parent" stream) be allocated resources before the former stream (the "dependent" stream). Taken together, the dependencies across all streams in a connection form a dependency tree. The structure of the dependency tree changes as HTTP/2 HEADERS and PRIORITY frames add, remove, or change the dependency links between streams.

Implicit in this scheme is the notion of in-order delivery of priority changes (i.e., dependency tree mutations): since operations on the dependency tree such as reparenting a subtree are not commutative, both sender and receiver must apply them in the same order to ensure that both sides have a consistent view of the stream dependency tree. HTTP/2 specifies priority assignments in PRIORITY frames and (optionally) in HEADERS frames. To achieve in-order delivery of HTTP/2 priority changes in HTTP/2-over-QUIC, HTTP/2 PRIORITY frames, in addition to HEADERS frames, are also sent on reserved stream 3. The semantics of the Stream Dependency, Weight, E flag, and (for HEADERS frames) PRIORITY flag are the same as in HTTP/2-over-TCP.

Since HEADERS and PRIORITY frames are sent on a different stream than the STREAM frames for the streams they reference, they may be

delivered out-of-order with respect to the STREAM frames. There is no special handling for this--the receiver should simply assign resources according to the most recent stream priority information that it has received.

ALTERNATIVE DESIGN: if the core QUIC protocol implements priorities, then this document should map the HTTP/2 priorities scheme to that provided by the core protocol. This would likely involve prohibiting the sending of HTTP/2 PRIORITY frames and setting of the PRIORITY flag in HTTP/2 HEADERS frames, to avoid conflicting directives.

8. Flow Control

QUIC provides stream and connection level flow control, similar in principle to HTTP/2's flow control but with some implementation differences. As flow control is handled by QUIC, the HTTP/2 mapping need not concern itself with maintaining flow control state, or how/when to send flow control frames to the peer. The HTTP/2 mapping must not send HTTP/2 WINDOW_UPDATE frames.

The initial flow control window sizes (stream and connection) are communicated during the crypto handshake (see [Connection establishment]). Setting these values to the maximum size (2^31 - 1) effectively disables flow control.

Relatively small initial windows can be used, as QUIC will attempt to auto-tune the flow control windows based on usage. See [draft-hamilton-quic-transport-protocol] for more details.

9. Server Push

HTTP/2-over-QUIC supports HTTP/2 server push. During connection establishment, the client indicates whether or it is willing to receive server pushes via the SETTINGS_ENABLE_PUSH setting in the HTTP/2 SETTINGS frame (see [Connection Establishment]), which defaults to 1 (true).

As with server push for HTTP/2-over-TCP, the server initiates a server push by sending an HTTP/2 PUSH_PROMISE frame containing the StreamID of the stream to be pushed, as well as request header fields attributed to the request. The PUSH_PROMISE frame is sent on stream 3, to ensure proper ordering with respect to other HEADERS and non-data frames. Within the PUSH_PROMISE frame, the StreamID in the common HTTP/2 frame header indicates the associated (client-initiated) stream for the new push stream, while the Promised Stream ID field specifies the StreamID of the new push stream.

The server push response is conveyed in the same way as a non-server-push response, with response headers and (if present) trailers carried by HTTP/2 HEADERS frames sent on reserved stream 3, and response body (if any) sent via QUIC stream frames on the stream specified in the corresponding PUSH_PROMISE frame.

10. Error Codes

The HTTP/2 error codes defined in [RFC7540 Section 7] map to QUIC error codes as follows:

- NO_ERROR (0x0)
  - Maps to QUIC_NO_ERROR
- PROTOCOL_ERROR (0x1)
  - No single mapping?
- INTERNAL_ERROR (0x2)
  - QUIC_INTERNAL_ERROR? (not currently defined in core protocol spec)
- FLOW_CONTROL_ERROR (0x3)
  - QUIC_FLOW_CONTROL_RECEIVED_TOO_MUCH_DATA? (not currently defined in core protocol spec)
- SETTINGS_TIMEOUT (0x4)
  - ? (depends on whether we support SETTINGS acks)
- STREAM_CLOSED (0x5)
  - QUIC_STREAM_DATA_AFTER_TERMINATION
- FRAME_SIZE_ERROR (0x6)
  - QUIC_INVALID_FRAME_DATA
- REFUSED_STREAM (0x7)
  - ?
- CANCEL (0x8)
  - ?

- o COMPRESSION_ERROR (0x9)
  - * QUIC_DECOMPRESSION_FAILURE (not currently defined in core spec)
- o CONNECT_ERROR (0xa)
  - * ? (depends whether we decide to support CONNECT)
- o ENHANCE_YOUR_CALM (0xb)
  - * ?
- o INADEQUATE_SECURITY (0xc)
  - * QUIC_HANDSHAKE_FAILED, QUIC_CRYPTO_NO_SUPPORT
- o HTTP_1_1_REQUIRED (0xd)

TODO: fill in missing error code mappings.

11. Other HTTP/2 frames

QUIC includes some features (e.g. flow control) which are also present in HTTP/2. In these cases the HTTP/2 mapping need not re-implement them. As a result some HTTP/2 frame types are not required when using QUIC, as they either are directly implemented in the QUIC layer, or their functionality is provided via other means. This section of the document describes these cases.

11.1. GOAWAY frame

QUIC has its own GOAWAY frame, and QUIC implementations may to expose the sending of a GOAWAY to the application. The semantics of sending a GOAWAY in QUIC are identical to HTTP/2: an endpoint sending a GOAWAY will continue processing open streams, but will not accept newly created streams.

QUIC's GOAWAY frame is described in detail in the [draft-hamilton-quic-transport-protocol].

11.2. PING frame

QUIC has its own PING frame, which is currently exposed to the application. QUIC clients send periodic PINGs to servers if there are no currently active data streams on the connection.

QUIC's PING frame is described in detail in the [draft-hamilton-quic-transport-protocol].

11.3. PADDING frame

There is no HTTP/2 padding in this mapping; padding is instead provided at the QUIC layer by including QUIC PADDING frames in a packet payload. An HTTP/2 over QUIC mapping should treat any HTTP/2 level padding as an error, to avoid any possibility of inconsistent flow control states between endpoints (e.g. client sends HTTP/2 padding, counts it against flow control, server ignores).

12. Normative References

[RFC2119] Bradner, S., "Key Words for use in RFCs to Indicate Requirement Levels", March 1997.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, "Hypertext Transfer Protocol Version 2 (HTTP/2)", May 2015.

[RFC7541] Peon, R. and H. Ruellan, "HPACK: Header Compression for HTTP/2", May 2015.

[draft-hamilton-quic-transport-protocol] Hamilton, R., Iyengar, J., Swett, I., and A. Wilk, "QUIC: A UDP-Based Multiplexed and Secure Transport", July 2016.

[draft-thomson-quic-tls] Thomson, M. and R. Hamilton, "Porting QUIC to TLS", March 2016.

[draft-iyengar-quic-loss-recovery] Iyengar, J. and I. Swett, "QUIC Loss Recovery and Congestion Control", July 2016.

Authors' Addresses

Robbie Shade
Google

Email: rjshade@google.com

Mike Warres
Google

Email: mpw@google.com

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 28, 2017

M. Thomson
Mozilla
R. Hamilton
Google

October 25, 2016

Using Transport Layer Security (TLS) to Secure QUIC
draft-thomson-quic-tls-01

Abstract

This document describes how Transport Layer Security (TLS) can be used to secure QUIC.

Status of This Memo

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

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

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

This Internet-Draft will expire on April 28, 2017.

Copyright Notice

Table of Contents

1. Introduction

QUIC [I-D.hamilton-quic-transport-protocol] provides a multiplexed transport for HTTP [RFC7230] semantics that provides several key advantages over HTTP/1.1 [RFC7230] or HTTP/2 [RFC7540] over TCP [RFC0793].

This document describes how QUIC can be secured using Transport Layer Security (TLS) version 1.3 [I-D.ietf-tls-tls13]. TLS 1.3 provides critical latency improvements for connection establishment over

previous versions. Absent packet loss, most new connections can be established and secured within a single round trip; on subsequent connections between the same client and server, the client can often send application data immediately, that is, zero round trip setup.

This document describes how the standardized TLS 1.3 can act a security component of QUIC. The same design could work for TLS 1.2, though few of the benefits QUIC provides would be realized due to the handshake latency in versions of TLS prior to 1.3.

1.1. Notational Conventions

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

2. Protocol Overview

QUIC [I-D.hamilton-quic-transport-protocol] can be separated into several modules:

1. The basic frame envelope describes the common packet layout. This layer includes connection identification, version negotiation, and includes markers that allow the framing and public reset to be identified.

2. The public reset is an unprotected packet that allows an intermediary (an entity that is not part of the security context) to request the termination of a QUIC connection.

3. Version negotiation frames are used to agree on a common version of QUIC to use.

4. Framing comprises most of the QUIC protocol. Framing provides a number of different types of frame, each with a specific purpose. Framing supports frames for both congestion management and stream multiplexing. Framing additionally provides a liveness testing capability (the PING frame).

5. Encryption provides confidentiality and integrity protection for frames. All frames are protected based on keying material derived from the TLS connection running on stream 1. Prior to this, data is protected with the 0-RTT keys.

6. Multiplexed streams are the primary payload of QUIC. These provide reliable, in-order delivery of data and are used to carry the encryption handshake and transport parameters (stream 1), HTTP header fields (stream 3), and HTTP requests and responses.

Frames for managing multiplexing include those for creating and destroying streams as well as flow control and priority frames.

7. Congestion management includes packet acknowledgment and other signal required to ensure effective use of available link capacity.

8. A complete TLS connection is run on stream 1. This includes the entire TLS record layer. As the TLS connection reaches certain states, keying material is provided to the QUIC encryption layer for protecting the remainder of the QUIC traffic.

9. HTTP mapping provides an adaptation to HTTP that is based on HTTP/2.

The relative relationship of these components are pictorally represented in Figure 1.

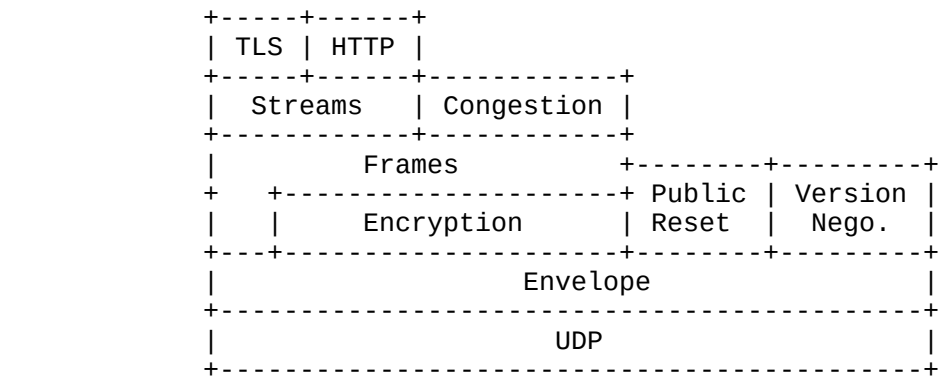


Figure 1: QUIC Structure

This document defines the cryptographic parts of QUIC. This includes the handshake messages that are exchanged on stream 1, plus the record protection that is used to encrypt and authenticate all other frames.

2.1. Handshake Overview

TLS 1.3 provides two basic handshake modes of interest to QUIC:

- A full handshake in which the client is able to send application data after one round trip and the server immediately after receiving the first message from the client.

- A 0-RTT handshake in which the client uses information about the server to send immediately. This data can be replayed by an attacker so it MUST NOT carry a self-contained trigger for any non-idempotent action.

A simplified TLS 1.3 handshake with 0-RTT application data is shown in Figure 2, see [I-D.ietf-tls-tls13] for more options and details.

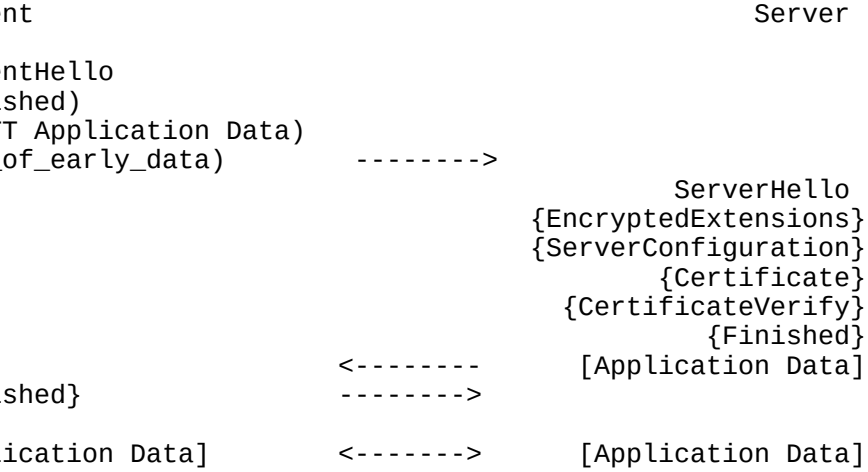


Figure 2: TLS Handshake with 0-RTT

Two additional variations on this basic handshake exchange are relevant to this document:

- The server can respond to a ClientHello with a HelloRetryRequest, which adds an additional round trip prior to the basic exchange. This is needed if the server wishes to request a different key exchange key from the client. HelloRetryRequest is also used to verify that the client is correctly able to receive packets on the address it claims to have (see Section 6.3).

- A pre-shared key mode can be used for subsequent handshakes to avoid public key operations. This is the basis for 0-RTT data, even if the remainder of the connection is protected by a new Diffie-Hellman exchange.

3. TLS in Stream 1

QUIC completes its cryptographic handshake on stream 1, which means that the negotiation of keying material happens after the QUIC protocol has started. This simplifies the use of TLS since QUIC is

able to ensure that the TLS handshake packets are delivered reliably and in order.

QUIC Stream 1 carries a complete TLS connection. This includes the TLS record layer in its entirety. QUIC provides for reliable and in-order delivery of the TLS handshake messages on this stream.

Prior to the completion of the TLS handshake, QUIC frames can be exchanged. However, these frames are not authenticated or confidentiality protected. Section 5 covers some of the implications of this design and limitations on QUIC operation during this phase.

Once complete, QUIC frames are protected using QUIC record protection, see Section 4.

3.1. Handshake and Setup Sequence

The integration of QUIC with a TLS handshake is shown in more detail in Figure 3. QUIC "STREAM" frames on stream 1 carry the TLS handshake. QUIC is responsible for ensuring that the handshake packets are re-sent in case of loss and that they can be ordered correctly.

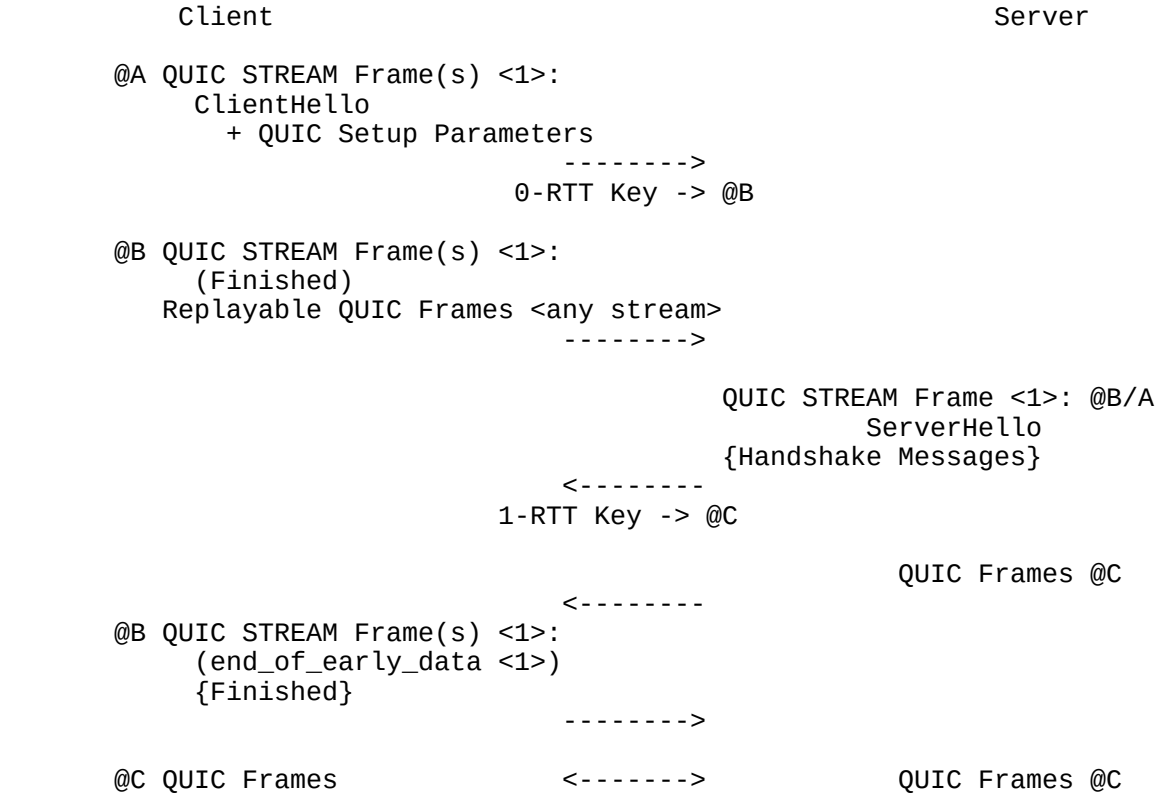


Figure 3: QUIC over TLS Handshake

In Figure 3, symbols mean:

- "<" and ">" enclose stream numbers.
- "@" indicates the key phase that is currently used for protecting QUIC packets.
- "(" and ")" enclose messages that are protected with TLS 0-RTT handshake or application keys.
- "{" and "}" enclose messages that are protected by the TLS Handshake keys.

If 0-RTT is not possible, then the client does not send frames protected by the 0-RTT key (@B). The only key transition on the client is from cleartext (@A) to 1-RTT protection (@C).

If 0-RTT data is not accepted by the server, then the server sends its handshake messages without protection (@A). The client still transitions from @A to @B, but it can stop sending 0-RTT data and progress immediately to 1-RTT data when it receives a cleartext ServerHello.

4. QUIC Record Protection

QUIC provides a record protection layer that is responsible for authenticated encryption of packets. The record protection layer uses keys provided by the TLS connection and authenticated encryption to provide confidentiality and integrity protection for the content of packets.

Different keys are used for QUIC and TLS record protection. Having separate QUIC and TLS record protection means that TLS records can be protected by two different keys. This redundancy is maintained for the sake of simplicity.

4.1. Key Phases

The transition to use of a new QUIC key occurs immediately after sending the TLS handshake messages that produced the key transition. Every time that a new set of keys is used for protecting outbound messages, the KEY_PHASE bit in the public flags is toggled. The KEY_PHASE bit on unencrypted messages is 0.

The KEY_PHASE bit on the public flags is the most significant bit (0x80).

The KEY_PHASE bit allows a recipient to detect a change in keying material without needing to receive the message that triggers the change. This avoids head-of-line blocking around transitions between keys without relying on trial decryption.

The following transitions are defined:

- The client transitions to using 0-RTT keys after sending the ClientHello. This causes the KEY_PHASE bit on packets sent by the client to be set to 1.

- The server transitions to using 0-RTT keys before sending the ServerHello, but only if the early data from the client is accepted. This transition causes the KEY_PHASE bit on packets sent by the server to be set to 1. If the server rejects 0-RTT data, the server’s handshake messages are sent without QUIC-level record protection with a KEY_PHASE of 0. TLS handshake messages

will still be protected by TLS record protection based on the TLS handshake traffic keys.

- The server transitions to using 1-RTT keys after sending its Finished message. This causes the KEY_PHASE bit to be set to 0 if early data was accepted, and 1 if the server rejected early data.

- The client transitions to 1-RTT keys after sending its Finished message. Subsequent messages from the client will then have a KEY_PHASE of 0 if 0-RTT data was sent, and 1 otherwise.

- Both peers start sending messages protected by a new key immediately after sending a TLS KeyUpdate message. The value of the KEY_PHASE bit is changed each time.

At each point, both keying material (see Section 4.2) and the AEAD function used by TLS is interchanged with the values that are currently in use for protecting outbound packets. Once a change of keys has been made, packets with higher sequence numbers MUST use the new keying material until a newer set of keys (and AEAD) are used. The exception to this is that retransmissions of TLS handshake packets MUST use the keys that they were originally protected with.

Once a packet protected by a new key has been received, a recipient SHOULD retain the previous keys for a short period. Retaining old keys allows the recipient to decode reordered packets around a change in keys. Keys SHOULD be discarded when an endpoints has received all packets with sequence numbers lower than the lowest sequence number used for the new key, or when it determines that reordering of those packets is unlikely. 0-RTT keys SHOULD be retained until the handshake is complete.

The KEY_PHASE bit does not directly indicate which keys are in use. Depending on whether 0-RTT data was sent and accepted, packets protected with keys derived from the same secret might be marked with different KEY_PHASE values.

4.1.1. Retransmission of TLS Handshake Messages

TLS handshake messages need to be retransmitted with the same level of cryptographic protection that was originally used to protect them. Newer keys cannot be used to protect QUIC packets that carry TLS messages.

A client would be unable to decrypt retransmissions of a server’s handshake messages that are protected using the 1-RTT keys, since the calculation of the application data keys depends on the contents of the handshake messages.

This restriction means the creation of an exception to the requirement to always use new keys for sending once they are available. A server MUST mark the retransmitted handshake messages with the same KEY_PHASE as the original messages to allow a recipient to distinguish the messages.

4.1.2. Key Update

Once the TLS handshake is complete, the KEY_PHASE bit allows for the processing of messages without having to receive the TLS KeyUpdate message that triggers the key update. This allows endpoints to start using updated keys immediately without the concern that a lost KeyUpdate will cause their messages to be indecipherable to their peer..

An endpoint MUST NOT initiate more than one key update at a time. A new key update cannot be sent until the endpoint has received a matching KeyUpdate message from its peer; or, if the endpoint did not initiate the original key update, it has received an acknowledgment of its own KeyUpdate.

This ensures that there are at most two keys to distinguish between at any one time, for which the KEY_PHASE bit is sufficient.

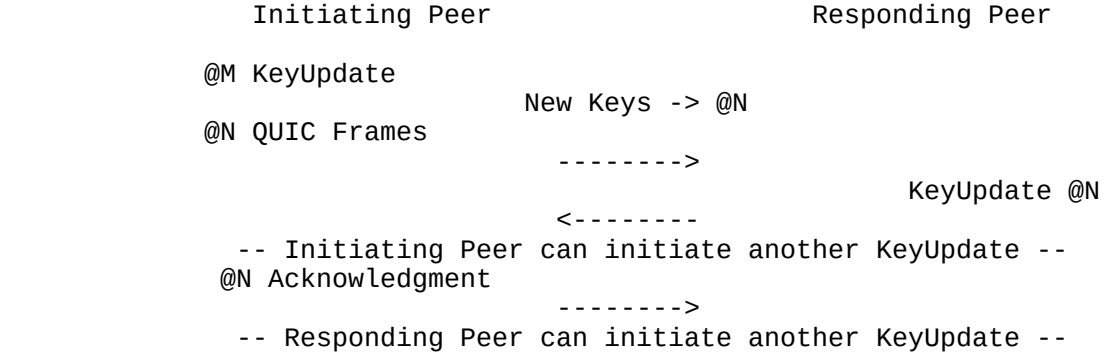


Figure 4: Key Update

As shown in Figure 3 and Figure 4, there is never a situation where there are more than two different sets of keying material that might be received by a peer.

A server cannot initiate a key update until it has received the client's Finished message. Otherwise, packets protected by the updated keys could be confused for retransmissions of handshake messages. A client cannot initiate a key update until it has

received an acknowledgment that its Finished message has been received.

Note: This models the key changes in the handshake as a key update initiated by the server, with the Finished message in the place of KeyUpdate.

4.2. QUIC Key Expansion

The following table shows QUIC keys, when they are generated and the TLS secret from which they are derived:

Key	TLS Secret	Phase
0-RTT	early_traffic_secret	"QUIC 0-RTT key expansion"
1-RTT	traffic_secret_N	"QUIC 1-RTT key expansion"

0-RTT keys are those keys that are used in resumed connections prior to the completion of the TLS handshake. Data sent using 0-RTT keys might be replayed and so has some restrictions on its use, see Section 5.2. 0-RTT keys are used after sending or receiving a ClientHello.

1-RTT keys are used after the TLS handshake completes. There are potentially multiple sets of 1-RTT keys; new 1-RTT keys are created by sending a TLS KeyUpdate message. 1-RTT keys are used after sending a Finished or KeyUpdate message.

The complete key expansion uses the same process for key expansion as defined in Section 7.3 of [I-D.ietf-tls-tls13]. For example, the Client Write Key for the data sent immediately after sending the TLS Finished message is:

```
label = "QUIC 1-RTT key expansion, client write key"
client_write = HKDF-Expand-Label(traffic_secret_0, label,
                                 "", key_length)
```

This results in a label input to HKDF that includes a two-octet length field, the string "TLS 1.3, QUIC 1-RTT key expansion, client write key" and a zero octet.

The QUIC record protection initially starts without keying material. When the TLS state machine produces the corresponding secret, new keys are generated from the TLS connection and used to protect the QUIC record protection.

The Authentication Encryption with Associated Data (AEAD) [RFC5116] function used is the same one that is negotiated for use with the TLS connection. For example, if TLS is using the TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256, the AEAD_AES_128_GCM function is used.

4.3. QUIC AEAD application

Regular QUIC packets are protected by an AEAD [RFC5116]. Version negotiation and public reset packets are not protected.

Once TLS has provided a key, the contents of regular QUIC packets immediately after any TLS messages have been sent are protected by the AEAD selected by TLS.

The key, K, for the AEAD is either the Client Write Key or the Server Write Key, derived as defined in Section 4.2.

The nonce, N, for the AEAD is formed by combining either the Client Write IV or Server Write IV with the sequence numbers. The 48 bits of the reconstructed QUIC sequence number (see Section 4.4) in network byte order is left-padded with zeros to the N_MAX parameter of the AEAD (see Section 4 of [RFC5116]). The exclusive OR of the padded sequence number and the IV forms the AEAD nonce.

The associated data, A, for the AEAD is an empty sequence.

The input plaintext, P, for the AEAD is the contents of the QUIC frame following the packet number, as described in [I-D.hamilton-quic-transport-protocol]

The output ciphertext, C, of the AEAD is transmitted in place of P.

Prior to TLS providing keys, no record protection is performed and the plaintext, P, is transmitted unmodified.

Note: QUIC defined a null-encryption that had an additional, hash-based checksum for cleartext packets. This might be added here, but it is more complex.

4.4. Sequence Number Reconstruction

Each peer maintains a 48-bit sequence number that is incremented with every packet that is sent, including retransmissions. The least significant 8-, 16-, 32-, or 48-bits of this number is encoded in the QUIC sequence number field in every packet.

A receiver maintains the same values, but recovers values based on the packets it receives. This is based on the sequence number of packets that it has received. A simple scheme predicts the receive sequence number of an incoming packet by incrementing the sequence number of the most recent packet to be successfully decrypted by one and expecting the sequence number to be within a range centered on that value.

A more sophisticated algorithm can almost double the search space by checking backwards from the most recent sequence for a received (or abandoned) packet. If a packet was received, then the packet contains a sequence number that is greater than the most recent sequence number. If no such packet was found, the number is assumed to be in the smaller window centered on the next sequence number, as in the simpler scheme.

Note: QUIC has a single, contiguous sequence number space. In comparison, TLS restarts its sequence number each time that record protection keys are changed. The sequence number restart in TLS ensures that a compromise of the current traffic keys does not allow an attacker to truncate the data that is sent after a key update by sending additional packets under the old key (causing new packets to be discarded). QUIC does not assume a reliable transport and is therefore required to handle attacks where packets are dropped in other ways. TLS maintains a separate sequence number that is used for record protection on the connection that is hosted on stream 1. This sequence number is reset according to the rules in the TLS protocol.

5. Pre-handshake QUIC Messages

Implementations MUST NOT exchange data on any stream other than stream 1 prior to the completion of the TLS handshake. However, QUIC requires the use of several types of frame for managing loss detection and recovery. In addition, it might be useful to use the data acquired during the exchange of unauthenticated messages for congestion management.

This section generally only applies to TLS handshake messages from both peers and acknowledgments of the packets carrying those messages. In many cases, the need for servers to provide acknowledgments is minimal, since the messages that clients send are small and implicitly acknowledged by the server’s responses.

The actions that a peer takes as a result of receiving an unauthenticated packet needs to be limited. In particular, state established by these packets cannot be retained once record protection commences.

There are several approaches possible for dealing with unauthenticated packets prior to handshake completion:

- discard and ignore them
- use them, but reset any state that is established once the handshake completes
- use them and authenticate them afterwards; failing the handshake if they can't be authenticated
- save them and use them when they can be properly authenticated
- treat them as a fatal error

Different strategies are appropriate for different types of data. This document proposes that all strategies are possible depending on the type of message.

- Transport parameters and options are made usable and authenticated as part of the TLS handshake (see Section 6.2).
- Most unprotected messages are treated as fatal errors when received except for the small number necessary to permit the handshake to complete (see Section 5.1).
- Protected packets can either be discarded or saved and later used (see Section 5.3).

5.1. Unprotected Frames Prior to Handshake Completion

This section describes the handling of messages that are sent and received prior to the completion of the TLS handshake.

Sending and receiving unprotected messages is hazardous. Unless expressly permitted, receipt of an unprotected message of any kind MUST be treated as a fatal error.

5.1.1. STREAM Frames

"STREAM" frames for stream 1 are permitted. These carry the TLS handshake messages.

Receiving unprotected "STREAM" frames for other streams MUST be treated as a fatal error.

5.1.2. ACK Frames

"ACK" frames are permitted prior to the handshake being complete. Information learned from "ACK" frames cannot be entirely relied upon, since an attacker is able to inject these packets. Timing and packet retransmission information from "ACK" frames is critical to the functioning of the protocol, but these frames might be spoofed or altered.

Endpoints MUST NOT use an unprotected "ACK" frame to acknowledge data that was protected by 0-RTT or 1-RTT keys. An endpoint MUST ignore an unprotected "ACK" frame if it claims to acknowledge data that was protected data. Such an acknowledgement can only serve as a denial of service, since an endpoint that can read protected data is always permitted to send protected data.

An endpoint SHOULD use data from unprotected or 0-RTT-protected "ACK" frames only during the initial handshake and while they have insufficient information from 1-RTT-protected "ACK" frames. Once sufficient information has been obtained from protected messages, information obtained from less reliable sources can be discarded.

5.1.3. WINDOW_UPDATE Frames

"WINDOW_UPDATE" frames MUST NOT be sent unprotected.

Though data is exchanged on stream 1, the initial flow control window is is sufficiently large to allow the TLS handshake to complete. This limits the maximum size of the TLS handshake and would prevent a server or client from using an abnormally large certificate chain.

Stream 1 is exempt from the connection-level flow control window.

5.1.4. Denial of Service with Unprotected Packets

Accepting unprotected - specifically unauthenticated - packets presents a denial of service risk to endpoints. An attacker that is able to inject unprotected packets can cause a recipient to drop even protected packets with a matching sequence number. The spurious packet shadows the genuine packet, causing the genuine packet to be ignored as redundant.

Once the TLS handshake is complete, both peers MUST ignore unprotected packets. The handshake is complete when the server receives a client’s Finished message and when a client receives an acknowledgement that their Finished message was received. From that point onward, unprotected messages can be safely dropped. Note that

the client could retransmit its Finished message to the server, so the server cannot reject such a message.

Since only TLS handshake packets and acknowledgments are sent in the clear, an attacker is able to force implementations to rely on retransmission for packets that are lost or shadowed. Thus, an attacker that intends to deny service to an endpoint has to drop or shadow protected packets in order to ensure that their victim continues to accept unprotected packets. The ability to shadow packets means that an attacker does not need to be on path.

ISSUE: This would not be an issue if QUIC had a randomized starting sequence number. If we choose to randomize, we fix this problem and reduce the denial of service exposure to on-path attackers. The only possible problem is in authenticating the initial value, so that peers can be sure that they haven’t missed an initial message.

In addition to denying endpoints messages, an attacker to generate packets that cause no state change in a recipient. See Section 7.2 for a discussion of these risks.

To avoid receiving TLS packets that contain no useful data, a TLS implementation MUST reject empty TLS handshake records and any record that is not permitted by the TLS state machine. Any TLS application data or alerts - other than a single end_of_early_data at the appropriate time - that is received prior to the end of the handshake MUST be treated as a fatal error.

5.2. Use of 0-RTT Keys

If 0-RTT keys are available, the lack of replay protection means that restrictions on their use are necessary to avoid replay attacks on the protocol.

A client MUST only use 0-RTT keys to protect data that is idempotent. A client MAY wish to apply additional restrictions on what data it sends prior to the completion of the TLS handshake. A client otherwise treats 0-RTT keys as equivalent to 1-RTT keys.

A client that receives an indication that its 0-RTT data has been accepted by a server can send 0-RTT data until it receives all of the server’s handshake messages. A client SHOULD stop sending 0-RTT data if it receives an indication that 0-RTT data has been rejected. In addition to a ServerHello without an early_data extension, an unprotected handshake message with a KEY_PHASE bit set to 0 indicates that 0-RTT data has been rejected.

A client SHOULD send its end_of_early_data alert only after it has received all of the server's handshake messages. Alternatively phrased, a client is encouraged to use 0-RTT keys until 1-RTT keys become available. This prevents stalling of the connection and allows the client to send continuously.

A server MUST NOT use 0-RTT keys to protect anything other than TLS handshake messages. Servers therefore treat packets protected with 0-RTT keys as equivalent to unprotected packets in determining what is permissible to send. A server protects handshake messages using the 0-RTT key if it decides to accept a 0-RTT key. A server MUST still include the early_data extension in its ServerHello message.

This restriction prevents a server from responding to a request using frames protected by the 0-RTT keys. This ensures that all application data from the server are always protected with keys that have forward secrecy. However, this results in head-of-line blocking at the client because server responses cannot be decrypted until all the server's handshake messages are received by the client.

5.3. Protected Frames Prior to Handshake Completion

Due to reordering and loss, protected packets might be received by an endpoint before the final handshake messages are received. If these can be decrypted successfully, such packets MAY be stored and used once the handshake is complete.

Unless expressly permitted below, encrypted packets MUST NOT be used prior to completing the TLS handshake, in particular the receipt of a valid Finished message and any authentication of the peer. If packets are processed prior to completion of the handshake, an attacker might use the willingness of an implementation to use these packets to mount attacks.

TLS handshake messages are covered by record protection during the handshake, once key agreement has completed. This means that protected messages need to be decrypted to determine if they are TLS handshake messages or not. Similarly, "ACK" and "WINDOW_UPDATE" frames might be needed to successfully complete the TLS handshake.

Any timestamps present in "ACK" frames MUST be ignored rather than causing a fatal error. Timestamps on protected frames MAY be saved and used once the TLS handshake completes successfully.

An endpoint MAY save the last protected "WINDOW_UPDATE" frame it receives for each stream and apply the values once the TLS handshake completes. Failing to do this might result in temporary stalling of affected streams.

6. QUIC-Specific Additions to the TLS Handshake

QUIC uses the TLS handshake for more than just negotiation of cryptographic parameters. The TLS handshake validates protocol version selection, provides preliminary values for QUIC transport parameters, and allows a server to perform return routeability checks on clients.

6.1. Protocol and Version Negotiation

The QUIC version negotiation mechanism is used to negotiate the version of QUIC that is used prior to the completion of the handshake. However, this packet is not authenticated, enabling an active attacker to force a version downgrade.

To ensure that a QUIC version downgrade is not forced by an attacker, version information is copied into the TLS handshake, which provides integrity protection for the QUIC negotiation. This does not prevent version downgrade during the handshake, though it means that such a downgrade causes a handshake failure.

Protocols that use the QUIC transport MUST use Application Layer Protocol Negotiation (ALPN) [RFC7301]. The ALPN identifier for the protocol MUST be specific to the QUIC version that it operates over. When constructing a ClientHello, clients MUST include a list of all the ALPN identifiers that they support, regardless of whether the QUIC version that they have currently selected supports that protocol.

Servers SHOULD select an application protocol based solely on the information in the ClientHello, not using the QUIC version that the client has selected. If the protocol that is selected is not supported with the QUIC version that is in use, the server MUST either send a QUIC version negotiation packet if this is possible, or fail the connection otherwise.

6.2. QUIC Extension

QUIC defines an extension for use with TLS. That extension defines transport-related parameters. This provides integrity protection for these values. Including these in the TLS handshake also make the values that a client sets available to a server one-round trip earlier than parameters that are carried in QUIC frames. This document does not define that extension.

6.3. Source Address Validation

QUIC implementations describe a source address token. This is an opaque blob that a server might provide to clients when they first use a given source address. The client returns this token in subsequent messages as a return routeability check. That is, the client returns this token to prove that it is able to receive packets at the source address that it claims. This prevents the server from being used in packet reflection attacks (see Section 7.1).

A source address token is opaque and consumed only by the server. Therefore it can be included in the TLS 1.3 pre-shared key identifier for 0-RTT handshakes. Servers that use 0-RTT are advised to provide new pre-shared key identifiers after every handshake to avoid linkability of connections by passive observers. Clients MUST use a new pre-shared key identifier for every connection that they initiate; if no pre-shared key identifier is available, then resumption is not possible.

A server that is under load might include a source address token in the cookie extension of a HelloRetryRequest. (Note: the current version of TLS 1.3 does not include the ability to include a cookie in HelloRetryRequest.)

6.4. Priming 0-RTT

QUIC uses TLS without modification. Therefore, it is possible to use a pre-shared key that was obtained in a TLS connection over TCP to enable 0-RTT in QUIC. Similarly, QUIC can provide a pre-shared key that can be used to enable 0-RTT in TCP.

All the restrictions on the use of 0-RTT apply, and the certificate MUST be considered valid for both connections, which will use different protocol stacks and could use different port numbers. For instance, HTTP/1.1 and HTTP/2 operate over TLS and TCP, whereas QUIC operates over UDP.

Source address validation is not completely portable between different protocol stacks. Even if the source IP address remains constant, the port number is likely to be different. Packet reflection attacks are still possible in this situation, though the set of hosts that can initiate these attacks is greatly reduced. A server might choose to avoid source address validation for such a connection, or allow an increase to the amount of data that it sends toward the client without source validation.

7. Security Considerations

There are likely to be some real clangers here eventually, but the current set of issues is well captured in the relevant sections of the main text.

Never assume that because it isn't in the security considerations section it doesn't affect security. Most of this document does.

7.1. Packet Reflection Attack Mitigation

A small ClientHello that results in a large block of handshake messages from a server can be used in packet reflection attacks to amplify the traffic generated by an attacker.

Certificate caching [RFC7924] can reduce the size of the server's handshake messages significantly.

A client SHOULD also pad [RFC7685] its ClientHello to at least 1024 octets (TODO: tune this value). A server is less likely to generate a packet reflection attack if the data it sends is a small multiple of the data it receives. A server SHOULD use a HelloRetryRequest if the size of the handshake messages it sends is likely to exceed the size of the ClientHello.

7.2. Peer Denial of Service

QUIC, TLS and HTTP/2 all contain a 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. If processing is disproportionately large in comparison to the observable effects on bandwidth or state, then this could allow a malicious peer to exhaust processing capacity without consequence.

QUIC prohibits the sending of empty "STREAM" frames unless they are marked with the FIN bit. This prevents "STREAM" frames from being sent that only waste effort.

TLS records SHOULD always contain at least one octet of a handshake messages or alert. Records containing only padding are permitted during the handshake, but an excessive number might be used to generate unnecessary work. Once the TLS handshake is complete, endpoints SHOULD NOT send TLS application data records unless it is to hide the length of QUIC records. QUIC packet protection does not include any allowance for padding; padded TLS application data records can be used to mask the length of QUIC frames.

While there are legitimate uses for some redundant packets, implementations SHOULD track redundant packets and treat excessive volumes of any non-productive packets as indicative of an attack.

8. IANA Considerations

This document has no IANA actions. Yet.

9. References

9.1. Normative References

[I-D.hamilton-quic-transport-protocol]
Hamilton, R., Iyengar, J., Swett, I., and A. Wilk, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-hamilton-quic-transport-protocol-00 (work in progress), July 2016.

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-17 (work in progress), October 2016.

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

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

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

9.2. Informative References

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <http://www.rfc-editor.org/info/rfc793>.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <http://www.rfc-editor.org/info/rfc7230>.

[RFC7258] Farrell, S. and H. Tschofenig, "Pervasive Monitoring Is an Attack", BCP 188, RFC 7258, DOI 10.17487/RFC7258, May 2014, <http://www.rfc-editor.org/info/rfc7258>.

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

[RFC7685] Langley, A., "A Transport Layer Security (TLS) ClientHello Padding Extension", RFC 7685, DOI 10.17487/RFC7685, October 2015, <http://www.rfc-editor.org/info/rfc7685>.

[RFC7924] Santesson, S. and H. Tschofenig, "Transport Layer Security (TLS) Cached Information Extension", RFC 7924, DOI 10.17487/RFC7924, July 2016, <http://www.rfc-editor.org/info/rfc7924>.

Appendix A. Acknowledgments

Christian Huitema’s knowledge of QUIC is far better than my own. This would be even more inaccurate and useless if not for his assistance. This document has variously benefited from a long series of discussions with Jana Iyengar, Adam Langley, Roberto Peon, Eric Rescorla, Ian Swett, and likely many others who are merely forgotten by a faulty meat computer.

Authors’ Addresses

Martin Thomson
Mozilla

Email: martin.thomson@gmail.com

Ryan Hamilton
Google

Email: rch@google.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: May 17, 2018

M. Kuehlewind
B. Trammell
ETH Zurich
J. Hildebrand
November 13, 2017

Transport-Independent Path Layer State Management
draft-trammell-plus-statefulness-04

Abstract

This document describes a simple state machine for stateful network devices on a path between two endpoints to associate state with traffic traversing them on a per-flow basis, as well as abstract signaling mechanisms for driving the state machine. This state machine is intended to replace the de-facto use of the TCP state machine or incomplete forms thereof by stateful network devices in a transport-independent way, while still allowing for fast state timeout of non-established or undesirable flows.

Table of Contents

1. Introduction

The boundary between the network and transport layers was originally defined to be that between information used (and potentially modified) hop-by-hop, and that used end-to-end. End-to-end information in the transport layer is associated with state at the endpoints, but processing of network-layer information was assumed to be stateless.

The widespread deployment of stateful middleboxes in the Internet, such as network address and port translators (NAPT), firewalls that model the TCP state machine to distinguish packets belonging from desirable flows from backscatter and random attack traffic, and devices which keep per-flow state for reporting and monitoring

purposes (e.g. IPFIX [RFC7011] Metering Processes), has broken this assumption, and made it more difficult to deploy non-TCP transport protocols in the Internet.

The deployment of new transport protocols encapsulated in UDP with encrypted transport headers (such as QUIC [I-D.ietf-quic-transport]) will present a challenge to the operation of these devices, and their ubquity likewise threatens to impair the deployability of these protocols. There are two main causes for this problem: first, stateful devices often use an internal model of the TCP state machine to determine when TCP flows start and end, allowing them to manage state for these flows; for UDP flows, they must rely on timeouts. These timeouts are generally short relative to those for TCP [IMC-GATEWAYS], requiring UDP- encapsulated transports either to generate unproductive keepalive traffic for long-lived sessions, or to tolerate connectivity problems and the necessity of reconnection due to loss of on-path state.

This document presents an abstract solution to this problem by defining a transport-independent state machine to be implemented at per-flow state- keeping middleboxes as a replacement for incomplete TCP state modeling. A key concept behind this approach is that encryption of transport protocol headers allows a transport protocol to separate its wire image - what it looks like to devices on path - from its internal semantics. We advocate the creation of a minimal wire image for these protocols that exposes enough information to drive the state machine presented. Present and future evolution of encrypted transport protocols can then happen behind this wire image, and Middleboxes implementing this state machine can use signals from a UDP encapsulation common to a set of encrypted transport protocols can have equivalent state information to that provided by TCP, reducing the friction between deployed middleboxes and these new transport protocols.

2. Terminology

In this document, the term "flow" is defined to be compatible with the definition given in [RFC7011]: A flow is defined as a set of packets passing a device on the network during a certain time interval. All packets belonging to a particular Flow have a set of common properties. Each property is defined as the result of applying a function to the values of:

1. one or more network layer header fields (e.g., destination IP address) or transport layer header fields (e.g., destination port number) that the device has access to;

2. one or more characteristics of the packet itself (e.g., number of MPLS labels, etc.);

3. one or more of the fields derived from packet treatment at the device (e.g., next-hop IP address, the output interface, etc.).

A packet is defined as belonging to a flow if it completely satisfies all the defined properties of the flow.

A bidirectional flow or biflow is defined as compatible with [RFC5103], by joining the "forward direction" flow with the "reverse direction" flow, derived by reversing the direction of directional fields (ports and IP addresses). Biflows are only relevant at devices positioned so as to see all the packets in both directions of the biflow, generally on the endpoint side of the service demarcation point for either endpoint as defined in the reference path given in [RFC7398].

3. State Machine

A transport-independent state machine for on-path devices is shown in Figure 1. It was designed to have the following properties:

- A device on path that can see traffic in both directions between two endpoints knows that each side of an association wishes that association to continue. This allows firewalls to delegate policy decisions about accepting or continuing an association to the servers they protect.

- A device on path that can see traffic in both directions between two endpoints knows that each device can receive traffic at the source address it provides. This allows firewalls to provide protection against trivially spoofed packets.

Both of these properties hold with current firewalls and network address translation devices observing the flags and sequence/ acknowledgment numbers exposed by TCP.

It relies on six states, three configurable timeouts, and a set of signals defined in Section 4. The states are defined as follows:

- zero: there is no state for a given flow at the device

- uniflow: at least one packet has been seen in one direction

- associating: at least one packet has been seen in one direction, and an indication that the receiving endpoint wishes to continue the association has been seen in the other direction.

- associated: a flow in associating state has further demonstrated that the initial sender can receive packets at its given source address.

- stop-wait: one side of a connection has sent an explicit stop signal, waiting for confirmation

- stopping: stop signal confirmed, association is stopping.

We refer to the zero and uniflow states as "uniflow states", as they are relevant both for truly unidirectional flows, as well as in situations where an on-path device can see only one side of a communication. We refer to the remaining four states as "biflow states", as they are only applicable to true bidirectional flows, where the on-path device can see both sides of the communication.

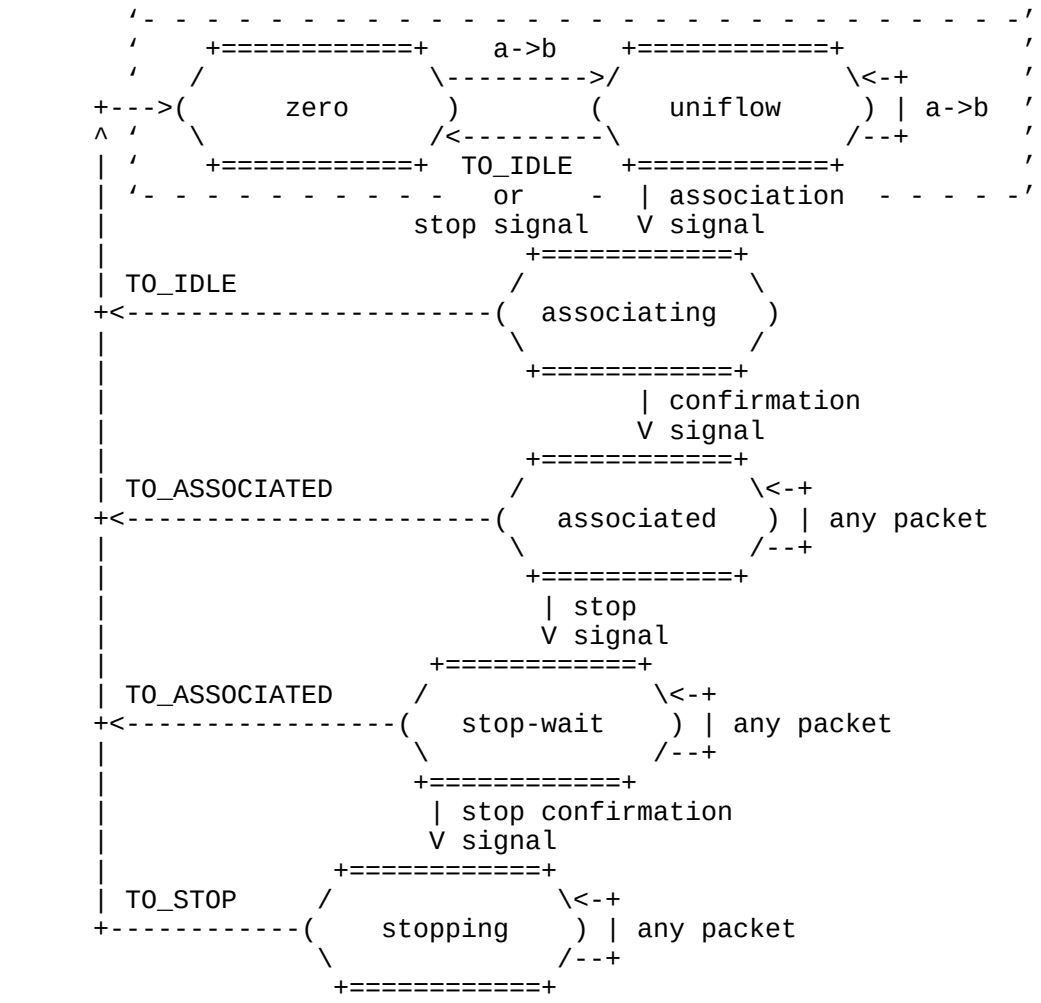


Figure 1: Transport-Independent State Machine for Stateful On-Path Devices

The three timeouts are defined as follows:

- TO_IDLE, the unidirectional idle timeout, can be considered equivalent to the idle timeout for transport protocols where the device has no information about session start and end (e.g. most UDP protocols).

- TO_ASSOCIATED, the bidirectional idle timeout, can be considered equivalent to the timeout for transport protocols where the device has information about session start and end (e.g. TCP).

- o TO_STOP is the teardown timeout: how long the device will account additional packets to a flow after confirming a close signal, ensuring retransmitted and/or reordered close signal don't lead to the spurious creation of new flow state.

Selection of timeouts is a configuration and implementation detail, but generally TO_STOP <= TO_IDLE << TO_ASSOCIATED; see [IMC-GATEWAYS] for an analysis of the magnitudes of these timeouts in presently deployed gateway devices.

3.1. Uniflow States

Every packet received by a device keeping per-flow state must associate that packet with a flow (see Section 4.1). When a device receives a packet associated with a flow it has no state for, and it is configured to forward the packet instead of dropping it, it moves that flow from the zero state into the uniflow state and starts a timer TO_IDLE. It resets this timer for any additional packet it forwards in the same direction as long as the flow remains in the uniflow state. When timer TO_IDLE expires on a flow in the uniflow state, the device drops state for the flow and performs any processing associated with doing so: tearing down NAT bindings, stopping associated firewall pinholes, exporting flow information, and so on. The device may also drop state on a stop signal, if observed.

Some devices will only see one side of a communication, e.g. if they are placed in a portion of a network with asymmetric routing. These devices use only the zero and uniflow states (as marked in Figure 1.) In addition, true uniflows - protocols which are solely unidirectional (e.g. some applications over UDP) - will also use only the uniflow-only states. In either case, current devices generally don't associate much state with observed uniflows, and an idle timeout is generally sufficient to expire this state.

3.2. Biflow States

A uniflow transitions to the associating state when the device observes an association signal, and further to the associated state when the device observes a subsequent confirmation signal; see Section 4.2 for details. If the flow has not transitioned to from the associating to the associated state after TO_IDLE, the device drops state for the flow.

After transitioning to the associated state, the device starts a timer TO_ASSOCIATED. It resets this timer for any packet it forwards in either direction. The associated state represents a fully established bidirectional communication. When timer TO_ASSOCIATED

expires, the device assumes that the flow has shut down without signaling as such, and drops state for the flow, performing any associated processing. When a bidirectional stop signal (see Section 4.3) is confirmed, the flow transitions to the stopping state.

When a flow enters the stopping state, it starts a timer TO_STOP. While the stop signal should be the last packet on a flow, the TO_STOP timer ensures that reordered packets after the stop signal will be accounted to the flow. When this timer expires, the device drops state for the flow, performing any associated processing.

3.3. Additional States and Actions

This document is concerned only with states and transitions common to transport- and function- independent state maintenance. Devices may augment the transitions in this state diagram depending on their function. For example, a firewall that decides based on some information beyond the signals used by this state machine to shut down a flow may transition it directly to a blacklist state on shutdown. Or, a firewall may fail to forward additional packets in the uniflow state until an association signal is observed.

4. Abstract Signaling Mechanisms

The state machine in Section 3 requires four signals: a new flow signal, the first packet observed in a flow in the zero state; an association signal, allowing a device to verify that an endpoint wishes a bidirectional communication to be established or to continue; a confirmation signal, allowing a device to confirm that the initiator of a flow is reachable at its purported source address; and a stop signal, noting that an endpoint wishes to stop a bidirectional communication. Additional related signals may also be useful, depending on the function a device provides. There are a few different ways to implement these signals; here, we explore the properties of some potential implementations.

We assume the following general requirements for these signals; parallel to those given in [draft-trammell-plus-abstract-mech]:

- At least the endpoints can verify the integrity of the signals exposed, and shut down a transport association when that verification fails, in order to reduce the incentive for on-path devices to attempt to spoof these signals.

- Endpoints and devices on path can probabilistically verify that a originator of a signal is on-path.

4.1. Flow Identification

In order to keep per-flow state, each device using this state machine must have a function it can apply to each packet to be able to extract common properties to identify the flow it is associated with. In general, the set of properties used for flow identification on presently deployed devices includes the source and destination IP address, the source and destination transport layer port number, the transport protocol number. The differentiated services field [RFC2474] may also be included in the set of properties defining a flow, since it may indicate different forwarding treatment.

However, other protocols may use additional bits in their own headers for flow identification. In any case, a protocol implementing signaling for this state machine must specify the function used for flow identification.

4.2. Association and Confirmation Signaling

An association signal indicates that the endpoint that received the first packet seen by the device has indeed seen that packet, and is interested in continuing conversation with the sending endpoint. This signal is roughly an in-band analogue to consent signaling in ICE [RFC7675] that is carried to every device along the path.

A confirmation signal indicates that the endpoint that sent the first packet seen by the device is reachable at its purported source address, and is necessary to prevent spoofed or reflected packets from driving the state machine into the associated state. It is roughly equivalent to the final ACK in the TCP three-way handshake.

These two signals are related to each other, in that association requires the receiving endpoint of the first packet to prove it has seen that packet (or a subsequent packet), and to acknowledge it wants to continue the association; while confirmation requires the sending endpoint to prove it has seen the association token.

Transport-independent, path-verifiable association and confirmation signaling can be implemented using three values carried in the packet headers: an association token, a confirmation nonce, and an echo token.

The association token is a cryptographically random value generated by the endpoint initiating a connection, and is carried on packets in the uniflow state. When a receiving endpoint wishes to send an association signal, it generates an echo token from the association token using a well-known, defined function (e.g. a truncated SHA-256 hash), and generates a cryptographically random confirmation nonce.

The initiating endpoint sends a confirmation signal on the next packet it sends after receiving the confirmation nonce, by applying a function to the echo token and the confirmation nonce, and sending the result as a new association token.

Devices on path verify that the echo token corresponds to a previously seen association token to recognize an association signal, and recognize that an association token corresponds to a previously seen echo token and confirmation nonce to recognize an association signal.

If the association token and confirmation nonce are predictable, off-path devices can spoof association and confirmation signals. In choosing the number of bits for an association token, there is a tradeoff between per-packet overhead and state overhead at on-path devices, and assurance that an association token is hard to guess. This tradeoff must be evaluated at protocol design time.

There are a few considerations in choosing a function (or functions) to generate the echo token from the association token, to verify an echo token given an association token, and to derive a next association token from the echo token and confirmation nonce. The functions could be extremely simple (e.g., identity for the echo token and addition for the nonce) for ease of implementation even in extremely constrained environments. Using one-way functions (e.g., truncated SHA-256 hash to derive echo token from association token; XOR followed by truncated SHA-256 hash to derive association token from echo token and confirmation nonce) requires slightly more work from on-path devices, but the primitives will be available at any endpoint using an encrypted transport protocol. In any case, a concrete implementation of association and confirmation signaling must choose a set of functions, or mechanism for unambiguously choosing one, at both endpoints as well as along the path.

4.2.1. Start-of-flow versus continual signaling

There are two possible points in the design space here: these signals could be continually exposed throughout the flow, or could be exposed only on the first few packets of a connection (those corresponding to the cryptographic and/or transport state handshakes in the overlying protocols).

In the former case, an on-path device could re-establish state in the middle of a flow; e.g. due to a reboot of the device, due to a NAT association change without the endpoints' knowledge, or due to idle periods longer than the TO_ESTABLISHED timeout value. The on-path device would receive no special information about which packets were associated with the start of association. In this case, the series

of exposed association tokens, echo tokens, and confirmation nonces can also be observed to derive a running round-trip time estimate for the flow.

In the latter case, an on-path device would need to observe the start of the flow to establish state, and would be able to distinguish connection-start packets from other packets.

4.3. Bidirectional Stop Signaling

The transport-independent state machine uses bidirectional stop signaling to tear down state. This requires a stop signal to be observed in one direction, and a stop confirmation signal to be observed in the other, to complete tearing down an association.

A stop signal is directly carried or otherwise encoded in the protocol header to indicate that a flow is ending, whether normally or abnormally, and that state associated with the flow should be torn down. Upon decoding a stop signal, a device on path should move the flow from uniflow state to zero, or from associated state to stop-wait state, to wait for a confirmation signal in the other direction. While in stop-wait state, state will be maintained until a timer set to TO_ASSOCIATED expires, with any packet forwarded in either direction reseting the timer.

A stop confirmation signal is directly carried or otherwise encoded in the protocol header to indicate that the endpoint receiving the stop signal confirms that the stop signal is valid. The stop confirmation signal contains some assurance that the far endpoint has seen the stop signal. When a stop confirmation signal is observed in the opposite direction from the stop signal, a device on path should move the flow from stop-wait state to stopping state. The flow will then remain in stopping state until a timer set to TO_STOP has expired, after which state for the flow will be dropped. The stopping timeout TO_STOP is intended to ensure that any packets reordered in delivery are accounted to the flow before state for it is dropped.

We assume the encoding of stop and stop confirmation signals into a packet header, as with all other signals, is integrity protected end-to-end. Stop signals, as association signals, could be forged by a single on-path device. However, unless a stop confirmation signal that can be associated with the stop signal is observed in the other direction, the flow remains in stop-wait state, during which state is maintained and packets continue to be forwarded in both directions. So this attack is of limited utility; an attacker wishing to inject state teardown would need to control at least one on-path device on

each side of a target device to spoof both stop and corresponding stop confirmation signals.

4.3.1. Authenticated Stop Signaling

Additionally, the stop and stop confirmation signals could be designed to authenticate themselves. Each endpoint could reveal a stop hash during the initial association, which is the result of a chosen cryptographic hash function applied to a stop token which that endpoint keeps secret. An endpoint wishing to end the association then reveals the stop token, which can be verified both by the far endpoint and devices on path which have cached the stop hash to be authentic. A stop confirmation signal additionally contains information derived from the initiating stop signal's stop token, as further assurance that the stop token was observed by the far endpoint.

4.4. Separate Utility

Although all of these signals are required to drive the state machine described by this document, note that association/confirmation and bidirectional stop signaling have separate utility. A transport protocol may expose the end of a flow without any proof of association or confirmation of return routability of the initiator. Alternately, the transport protocol could rely on short timeouts to clean up stale state on path, while exposing continuous association and confirmation signals to quickly reestablish state.

5. Deployment Considerations

The state machine defined in this document is most useful when implemented in a single instantiation (wire format for signals, and selection of functions for deriving values to be exposed and verified) by multiple transport protocols. It is intended for use with protocols that encrypt their transport- layer headers, and that are encapsulated within UDP, as is the case with QUIC [I-D.ietf-quic-transport]. Definition of that instantiation is out of scope for the present revision of this document.

The following subsections discuss incentives for deployment of this state machine both at middleboxes and at endpoints.

5.1. Middlebox Deployment

The state machine defined herein is designed to replace TCP state-tracking for firewalls and NAT devices. When encrypted transport protocols encapsulated in UDP adopt a set of signals and a wire format for those signals to drive this state machine, these

middleboxes could continue using TCP-like logic to handle those UDP flows. Recognizing the wire format used by those signals would allow these middleboxes to distinguish "UDP with an encrypted transport" from undifferentiated UDP, and to treat the former case more like TCP, providing longer timeouts for established flows, as well as stateful defense against spoofed or reflected garbage traffic.

5.2. Endpoint Deployment

An encrypted, UDP-encapsulated transport protocol has two primary incentives to expose these signals. First, allowing firewalls on networks that generally block UDP (about 3-5% of Internet-connected networks, depending on the study) to distinguish "UDP with an encrypted transport" traffic from other UDP traffic may result in less blocking of that traffic. Second, the difference between the timeouts TO_IDLE and TO_ASSOCIATED, as well as the continuous state establishment possible with some instantiations of the association and confirmation signals, would allow these transport protocols to send less unproductive keepalive traffic for long-lived, sparse flows.

While both of these advantages require middleboxes on path to recognize and use the signals driving this state machine, we note that content providers driving the deployment of this protocols are also operators of their own content provision networks, and that many of the benefits of encrypted- encapsulated transport firewalls will accrue to them, giving these content providers incentives to deploy both endpoints and middleboxes.

6. Signal mappings for transport protocols

We now show how this state machine can be driven by signals available in TCP and QUIC.

6.1. Signal mapping for TCP

A mapping of TCP flags to transitions in to the state machine in Section 3 shows how devices currently using a model of the TCP state machine can be converted to use this state machine.

TCP [RFC0793] provides start-of-flow association only. A packet with the SYN and ACK flags set in the absence of the FIN or RST flags, and an in-window acknowledgment number, is synonymous with the association signal. A packet with the ACK flag set in the absence of the FIN or RST flags after an initial SYN, and an in-window acknowledgment number, is synonymous with the confirmation signal. For a typical TCP flow:

1. The initial SYN places the flow into uniflow state,

2. The SYN-ACK sent in reply acts as a association signal and places the flow into associating state,

3. The ACK sent in reply acts as a confirmation signal and places the flow into associated state,

4. The final FIN is a stop signal, and

5. the ACK of the final FIN is a stop confirmation signal, moving the flow into stopping state.

Note that abormally closed flows (with RST) do not provide stop confirmation, and are therefore not provided for by this state machine. Due to TCP's support for half-closed flows, additional state modeling is necessary to extract a stop signal from the final FIN.

Note also that the association and stop signals derived from the TCP header are not integrity protected, and association and confirmation signals based on in-window ACK are not particularly resistant to off-path attacks [IMC-TCP]. The state machine is therefore more susceptible to manipulation when used with vanilla TCP as when with a transport protocol providing full integrity protection for its headers end-to-end.

6.2. Signal mapping for QUIC

QUIC [I-D.ietf-quic-transport] is a moving target; however, signals for driving this state machine are fundamentally compatible with the protocol's design and could easily be added to the protocol specification.

Specifically, QUIC's handshake is visible to on-path devices, as it begins with an unencrypted version negotiation which exposes a 64-bit connection ID, which can serve as an association and echo token as in Section 4.2. The function of the confirmation nonce is not fully exposed to the path at this point, but could be implemented by exposing information from the proof of source address ownership (section 7.4 of [I-D.ietf-quic-transport]) or via echoing the random initial packet number (as suggested by https://github.com/quicwg/base-drafts/pull/391).

The addition of a public reset signal that would act as a stop signal as in Section 4.3 is presently under discussion within the QUIC working group; the proposal for self-authenticating public reset at

https://github.com/quicwg/base-drafts/pull/20 inspired the addition of Section 4.3.1 to this document.

7. IANA Considerations

This document has no actions for IANA.

8. Security Considerations

This document defines a state machine for transport-independent state management on middleboxes, using in-band signaling, to replace the commonly- implemented current practice of incomplete TCP state modeling on these devices. It defines new signals for state management. While these signals can be spoofed by any device on path that observes traffic in both directions, we presume the presence of end-to-end integrity protection of these signals provided by the upper-layer transport driving them. This allows such spoofing to be detected and countered by endpoints, reducing the threat from on-path devices to connection disruption, which such devices are trivially placed to perform in any case.

9. Acknowledgments

Thanks to Christian Huitema for discussions leading to this document, and to Andrew Yourtchenko for the feedback. The mechanism for using a revealed value to prove ownership of a stop token was inspired by Eric Rescorla's suggestion to use a fundamentally identical mechanism for the QUIC public reset.

This work is partially supported by the European Commission under Horizon 2020 grant agreement no. 688421 Measurement and Architecture for a Middleboxed Internet (MAMI), and by the Swiss State Secretariat for Education, Research, and Innovation under contract no. 15.0268. This support does not imply endorsement.

10. References

10.1. Normative References

[RFC5103] Trammell, B. and E. Boschi, "Bidirectional Flow Export Using IP Flow Information Export (IPFIX)", RFC 5103, DOI 10.17487/RFC5103, January 2008, <https://www.rfc-editor.org/info/rfc5103>.

[RFC7011] Claise, B., Ed., Trammell, B., Ed., and P. Aitken, "Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information", STD 77, RFC 7011, DOI 10.17487/RFC7011, September 2013, <https://www.rfc-editor.org/info/rfc7011>.

[RFC7398] Bagnulo, M., Burbridge, T., Crawford, S., Eardley, P., and A. Morton, "A Reference Path and Measurement Points for Large-Scale Measurement of Broadband Performance", RFC 7398, DOI 10.17487/RFC7398, February 2015, <https://www.rfc-editor.org/info/rfc7398>.

10.2. Informative References

[draft-trammell-plus-abstract-mech]
Trammell, B., "Abstract Mechanisms for a Cooperative Path Layer under Endpoint Control", September 2016.

[I-D.hardie-path-signals]
Hardie, T., "Path signals", draft-hardie-path-signals-01 (work in progress), May 2017.

[I-D.ietf-quic-tls]
Thomson, M. and S. Turner, "Using Transport Layer Security (TLS) to Secure QUIC", draft-ietf-quic-tls-07 (work in progress), October 2017.

[I-D.ietf-quic-transport]
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-07 (work in progress), October 2017.

[IMC-GATEWAYS]
Hatonen, S., Nyrhinen, A., Eggert, L., Strowes, S., Sarolahti, P., and M. Kojo, "An experimental study of home gateway characteristics (Proc. ACM IMC 2010)", October 2010.

[IMC-TCP] Luckie, M., Beverly, R., Wu, T., Allman, M., and k. claffy, "Resilience of Deployed TCP to Blind Attacks. (Proc. ACM IMC 2015)", October 2015.

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <https://www.rfc-editor.org/info/rfc793>.

[RFC2474] Nichols, K., Blake, S., Baker, F., and D. Black, "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers", RFC 2474, DOI 10.17487/RFC2474, December 1998, <https://www.rfc-editor.org/info/rfc2474>.

[RFC7675] Perumal, M., Wing, D., Ravindranath, R., Reddy, T., and M. Thomson, "Session Traversal Utilities for NAT (STUN) Usage for Consent Freshness", RFC 7675, DOI 10.17487/RFC7675, October 2015, <https://www.rfc-editor.org/info/rfc7675>.

Authors' Addresses

Mirja Kuehlewind
ETH Zurich
Gloriastrasse 35
8092 Zurich
Switzerland

Email: mirja.kuehlewind@tik.ee.ethz.ch

Brian Trammell
ETH Zurich
Gloriastrasse 35
8092 Zurich
Switzerland

Email: ietf@trammell.ch

Joe Hildebrand

Email: hildjj@cursive.net