Network Working Group
Internet-Draft
Intended status: Informational
Expires: 23 October 2021

M. Kuehlewind
Ericsson
B. Trammell
Google

21 April 2021

Applicability of the QUIC Transport Protocol
draft-ietf-quic-applicability-11

Abstract

This document discusses the applicability of the QUIC transport protocol, focusing on caveats impacting application protocol development and deployment over QUIC. Its intended audience is designers of application protocol mappings to QUIC, and implementors of these application protocols.

Status of This Memo

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

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

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

This Internet-Draft will expire on 23 October 2021.

Table of Contents

1. Introduction

QUIC [QUIC] is a new transport protocol providing a number of advanced features. While initially designed for the HTTP use case, it provides capabilities that can be used with a much wider variety of applications. QUIC is encapsulated in UDP. QUIC version 1 integrates TLS 1.3 [TLS13] to encrypt all payload data and most control information. The version of HTTP that uses QUIC is known as HTTP/3 [QUIC-HTTP].

This document provides guidance for application developers that want to use the QUIC protocol without implementing it on their own. This includes general guidance for applications operating over HTTP/3 or directly over QUIC.

In the following sections we discuss specific caveats to QUIC's applicability, and issues that application developers must consider when using QUIC as a transport for their application.

2. The Necessity of Fallback

QUIC uses UDP as a substrate. This enables userspace implementation and permits traversal of network middleboxes (including NAT) without requiring updates to existing network infrastructure.

While recent measurements have shown no evidence of a widespread, systematic disadvantage of UDP traffic compared to TCP in the Internet [Edeline16], somewhere between three [Trammell16] and five [Swett16] percent of networks block all UDP traffic. All applications running on top of QUIC must therefore either be prepared to accept connectivity failure on such networks or be engineered to fall back to some other transport protocol. In the case of HTTP, this fallback is TLS over TCP.

The IETF TAPS specifications [I-D.ietf-taps-arch] describe a system with a common API for multiple protocols and some of the implications of fallback between these different protocols, specifically precluding fallback to insecure protocols or to weaker versions of secure protocols.

An application that implements fallback needs to consider the security consequences. A fallback to TCP and TLS exposes control information to modification and manipulation in the network. Further downgrades to older TLS versions than used in QUIC, which is 1.3, might result in significantly weaker cryptographic protection. For example, the results of protocol negotiation [RFC7301] only have confidentiality protection if TLS 1.3 is used.

These applications must operate, perhaps with impaired functionality, in the absence of features provided by QUIC not present in the fallback protocol. For fallback to TLS over TCP, the most obvious difference is that TCP does not provide stream multiplexing and therefore stream multiplexing would need to be implemented in the application layer if needed. Further, TCP implementations and network paths often do not support the Fast Open option [RFC7413], which enables sending of payload data together with the first control packet of a new connection as also provided by 0-RTT session resumption in QUIC. Note that there is some evidence of middleboxes

blocking SYN data even if TFO was successfully negotiated (see [PaaschNanog]). And even if Fast Open successfully operates end-to-end, it is limited to a single packet of TLS handshake and application data, unlike QUIC 0-RTT.

Moreover, while encryption (in this case TLS) is inseparably integrated with QUIC, TLS negotiation over TCP can be blocked. If TLS over TCP cannot be supported, the connection should be aborted, and the application then ought to present a suitable prompt to the user that secure communication is unavailable.

In summary, any fallback mechanism is likely to impose a degradation of performance and can degrade security; however, fallback must not silently violate the application's expectation of confidentiality or integrity of its payload data.

3. Zero RTT

QUIC provides for 0-RTT connection establishment. Though the same facility exists in TLS 1.3 with TCP, 0-RTT presents opportunities and challenges for applications using QUIC.

A transport protocol that provides 0-RTT connection establishment is qualitatively different than one that does not from the point of view of the application using it. Relative trade-offs between the cost of closing and reopening a connection and trying to keep it open are different; see Section 3.2.

An application needs to deliberately choose to use 0-RTT, as 0-RTT carries a risk of replay attack. Application protocols that use 0-RTT require a profile that describes the types of information that can be safely sent. For HTTP, this profile is described in [HTTP-REPLAY].

3.1. Replay Attacks

Retransmission or (malicious) replay of data contained in 0-RTT packets could cause the server side to receive two copies of the same data.

Application data sent by the client in 0-RTT packets could be processed more than once if it is replayed. Applications need to be aware of what is safe to send in 0-RTT. Application protocols that seek to enable the use of 0-RTT need a careful analysis and a description of what can be sent in 0-RTT; see Section 5.6 of [QUIC-TLS].

In some cases, it might be sufficient to limit application data sent in 0-RTT to that which only causes actions at a server that are known to be free of lasting effect. Initiating data retrieval or establishing configuration are examples of actions that could be safe. Idempotent operations - those for which repetition has the same net effect as a single operation - might be safe. However, it is also possible to combine individually idempotent operations into a non-idempotent sequence of operations.

Once a server accepts 0-RTT data there is no means of selectively discarding data that is received. However, protocols can define ways to reject individual actions that might be unsafe if replayed.

Some TLS implementations and deployments might be able to provide partial or even complete replay protection, which could be used to manage replay risk.

3.2. Session resumption versus Keep-alive

Because QUIC is encapsulated in UDP, applications using QUIC must deal with short network idle timeouts. Deployed stateful middleboxes will generally establish state for UDP flows on the first packet sent, and keep state for much shorter idle periods than for TCP. [RFC5382] suggests a TCP idle period of at least 124 minutes, though there is not evidence of widespread implementation of this guideline in the literature. Short network timeout for UDP, however, is well-documented. According to a 2010 study ([Hatonen10]), UDP applications can assume that any NAT binding or other state entry can expire after just thirty seconds of inactivity. Section 3.5 of [RFC8085] further discusses keep-alive intervals for UDP: it requires a minimum value of 15 seconds, but recommends larger values, or omitting keep-alive entirely.

By using a connection ID, QUIC is designed to be robust to NAT address rebinding after a timeout. However, this only helps if one endpoint maintains availability at the address its peer uses, and the peer is the one to send after the timeout occurs.

Some QUIC connections might not be robust to NAT rebinding because the routing infrastructure (in particular, load balancers) uses the address/port four-tuple to direct traffic. Furthermore, middleboxes with functions other than address translation could still affect the path. In particular, some firewalls do not admit server traffic for which the firewall has no recent state for a corresponding packet sent from the client.

QUIC applications can adjust idle periods to manage the risk of timeout. Idle periods and the network idle timeout are distinct from the connection idle timeout, which is defined as the minimum of either endpoint's idle timeout parameter; see Section 10.1 of [QUIC]). There are three options:

* Ignore the issue, if the application-layer protocol consists only of interactions with no or very short idle periods, or the protocol's resistance to NAT rebinding is sufficient.

* Ensure there are no long idle periods.

* Resume the session after a long idle period, using 0-RTT resumption when appropriate.

The first strategy is the easiest, but it only applies to certain applications.

Either the server or the client in a QUIC application can send PING frames as keep-alives, to prevent the connection and any on-path state from timing out. Recommendations for the use of keep-alives are application-specific, mainly depending on the latency requirements and message frequency of the application. In this case, the application mapping must specify whether the client or server is responsible for keeping the application alive. While [Hatonen10] suggests that 30 seconds might be a suitable value for the public Internet when a NAT is on path, larger values are preferable if the deployment can consistently survive NAT rebinding or is known to be in a controlled environment (e.g. data centres) in order to lower network and computational load.

Sending PING frames more frequently than every 30 seconds over long idle periods may result in excessive unproductive traffic in some situations, and to unacceptable power usage for power-constrained (mobile) devices. Additionally, timeouts shorter than 30 seconds can make it harder to handle transient network interruptions, such as VM migration or coverage loss during mobilty. See [RFC8085], especially Section 3.5.

Alternatively, the client (but not the server) can use session resumption instead of sending keepalive traffic. In this case, a client that wants to send data to a server over a connection idle longer than the server's idle timeout (available from the idle_timeout transport parameter) can simply reconnect. When possible, this reconnection can use 0-RTT session resumption, reducing the latency involved with restarting the connection. Of course, this approach is only valid in cases in which 0-RTT data is safe, when the client is the restarting peer, and when the data to be

sent is idempotent. It is also not applicable when the application binds external state to the connection, as this state cannot reliably be transferred to a resumed connection.

The tradeoffs between resumption and keep-alives need to be evaluated on a per-application basis. In general, applications should use keep-alives only in circumstances where continued communication is highly likely; [QUIC-HTTP], for instance, recommends using keep-alives only when a request is outstanding.

4. Use of Streams

QUIC's stream multiplexing feature allows applications to run multiple streams over a single connection, without head-of-line blocking between streams, associated at a point in time with a single five-tuple. Stream data is carried within frames, where one QUIC packet on the wire can carry one or multiple stream frames.

Streams can be unidirectional or bidirectional, and a stream may be initiated either by client or server. Only the initiator of a unidirectional stream can send data on it.

Streams and connections can each carry a maximum of 2^62-1 bytes in each direction, due to encoding limitations on stream offsets and connection flow control limits. In the presently unlikely event that this limit is reached by an application, a new connection would need to be established.

Streams can be independently opened and closed, gracefully or abruptly. An application can gracefully close the egress direction of a stream by instructing QUIC to send a FIN bit in a STREAM frame. It cannot gracefully close the ingress direction without a peer-generated FIN, much like in TCP. However, an endpoint can abruptly close the egress direction or request that its peer abruptly close the ingress direction; these actions are fully independent of each other.

QUIC does not provide an interface for exceptional handling of any stream. If a stream that is critical for an application is closed, the application can generate error messages on the application layer to inform the other end and/or the higher layer, which can eventually terminate the QUIC connection.

Mapping of application data to streams is application-specific and described for HTTP/3 in [QUIC-HTTP]. There are a few general principles to apply when designing an application's use of streams:

* A single stream provides ordering. If the application requires certain data to be received in order, that data should be sent on the same stream. There is no guarantee of transmission, reception, or delivery order across streams.

* Multiple streams provide concurrency. Data that can be processed independently, and therefore would suffer from head of line blocking if forced to be received in order, should be transmitted over separate streams.

* Streams can provide message orientation, and allow messages to be cancelled. If one message is mapped to a single stream, resetting the stream to expire an unacknowledged message can be used to emulate partial reliability for that message.

If a QUIC receiver has opened the maximum allowed concurrent streams, and the sender indicates that more streams are needed, it does not automatically lead to an increase of the maximum number of streams by the receiver. Therefore, an application can use the maximum number of allowed, currently open, and currently used streams when determining how to map data to streams.

QUIC assigns a numerical identifier to each stream, called the Stream ID. While the relationship between these identifiers and stream types is clearly defined in version 1 of QUIC, future versions might change this relationship for various reasons. QUIC implementations should expose the properties of each stream (which endpoint initiated the stream, whether the stream is unidirectional or bidirectional, the Stream ID used for the stream); applications should query for these properties rather than attempting to infer them from the Stream ID.

The method of allocating stream identifiers to streams opened by the application might vary between transport implementations. Therefore, an application should not assume a particular stream ID will be assigned to a stream that has not yet been allocated. For example, HTTP/3 uses Stream IDs to refer to streams that have already been opened, but makes no assumptions about future Stream IDs or the way in which they are assigned Section 6 of [QUIC-HTTP]).

4.1. Stream versus Flow Multiplexing

Streams are meaningful only to the application; since stream information is carried inside QUIC's encryption boundary, no information about the stream(s) whose frames are carried by a given packet is visible to the network. Therefore stream multiplexing is not intended to be used for differentiating streams in terms of network treatment. Application traffic requiring different network

treatment should therefore be carried over different five-tuples (i.e. multiple QUIC connections). Given QUIC's ability to send application data in the first RTT of a connection (if a previous connection to the same host has been successfully established to provide the necessary credentials), the cost of establishing another connection is extremely low.

4.2. Prioritization

Stream prioritization is not exposed to either the network or the receiver. Prioritization is managed by the sender, and the QUIC transport should provide an interface for applications to prioritize streams [QUIC]. Applications can implement their own prioritization scheme on top of QUIC: an application protocol that runs on top of QUIC can define explicit messages for signaling priority, such as those defined in [I-D.draft-ietf-httpbis-priority] for HTTP; it can define rules that allow an endpoint to determine priority based on context; or it can provide a higher level interface and leave the determination to the application on top.

Priority handling of retransmissions can be implemented by the sender in the transport layer. [QUIC] recommends retransmitting lost data before new data, unless indicated differently by the application. Currently, QUIC only provides fully reliable stream transmission, which means that prioritization of retransmissions will be beneficial in most cases, by filling in gaps and freeing up the flow control window. For partially reliable or unreliable streams, priority scheduling of retransmissions over data of higher-priority streams might not be desirable. For such streams, QUIC could either provide an explicit interface to control prioritization, or derive the prioritization decision from the reliability level of the stream.

4.3. Ordered and Reliable Delivery

QUIC streams enable ordered and reliable delivery. Though it is possible for an implementation to provide options that use streams for partial reliability or out-of-order delivery, most implementations will assume that data is reliably delivered in order.

Under this assumption, an endpoint that receives stream data might not make forward progress until data that is contiguous with the start of a stream is available. In particular, a receiver might withhold flow control credit until contiguous data is delivered to the application; see Section 2.2 of [QUIC]. To support this receive logic, an endpoint will send stream data until it is acknowledged, ensuring that data at the start of the stream is sent and acknowledged first.

An endpoint that uses a different sending behavior and does not negotiate that change with its peer might encounter performance issues or deadlocks.

4.4. Flow Control Deadlocks

QUIC flow control provides a means of managing access to the limited buffers endpoints have for incoming data. This mechanism limits the amount of data that can be in buffers in endpoints or in transit on the network. However, there are several ways in which limits can produce conditions that can cause a connection to either perform suboptimally or deadlock.

Deadlocks in flow control are possible for any protocol that uses QUIC, though whether they become a problem depends on how implementations consume data and provide flow control credit. Understanding what causes deadlocking might help implementations avoid deadlocks.

The size and rate of transport flow control credit updates can affect performance. Applications that use QUIC often have a data consumer that reads data from transport buffers. Some implementations might have independent transport-layer and application-layer receive buffers. Consuming data does not always imply it is immediately processed. However, a common flow control implementation technique is to extend credit to the sender, by emitting MAX_DATA and/or MAX_STREAM_DATA frames, as data is consumed. Delivery of these frames is affected by the latency of the back channel from the receiver to the data sender. If credit is not extended in a timely manner, the sending application can be blocked, effectively throttling the sender.

Large application messages can produce deadlocking if the recipient does not read data from the transport incrementally. If the message is larger than the flow control credit available and the recipient does not release additional flow control credit until the entire message is received and delivered, a deadlock can occur. This is possible even where stream flow control limits are not reached because connection flow control limits can be consumed by other streams.

A length-prefixed message format makes it easier for a data consumer to leave data unread in the transport buffer and thereby withhold flow control credit. If flow control limits prevent the remainder of a message from being sent, a deadlock will result. A length prefix might also enable the detection of this sort of deadlock. Where application protocols have messages that might be processed as a single unit, reserving flow control credit for the entire message atomically makes this style of deadlock less likely.

A data consumer can eagerly read all data as it becomes available, in order to make the receiver extend flow control credit and reduce the chances of a deadlock. However, such a data consumer might need other means for holding a peer accountable for the additional state it keeps for partially processed messages.

Deadlocking can also occur if data on different streams is interdependent. Suppose that data on one stream arrives before the data on a second stream on which it depends. A deadlock can occur if the first stream is left unread, preventing the receiver from extending flow control credit for the second stream. To reduce the likelihood of deadlock for interdependent data, the sender should ensure that dependent data is not sent until the data it depends on has been accounted for in both stream- and connection- level flow control credit.

Some deadlocking scenarios might be resolved by cancelling affected streams with STOP_SENDING or RESET_STREAM. Cancelling some streams results in the connection being terminated in some protocols.

4.5. Stream Limit Commitments

QUIC endpoints are responsible for communicating the cumulative limit of streams they would allow to be opened by their peer. Initial limits are advertised using the initial_max_streams_bidi and initial_max_streams_uni transport parameters. As streams are opened and closed they are consumed and the cumulative total is incremented. Limits can be increased using the MAX_STREAMS frame but there is no mechanism to reduce limits. Once stream limits are reached, no more streams can be opened, which prevents applications using QUIC from making further progress. At this stage connections can be terminated via idle timeout or explicit close; see Section 10).

An application that uses QUIC might communicate a cumulative stream limit but require the connection to be closed before the limit is reached. For example, to stop the server to perform scheduled maintenance. Immediate connection close causes abrupt closure of actively used streams. Depending on how an application uses QUIC streams, this could be undesirable or detrimental to behavior or

performance. A more graceful closure technique is to stop sending increases to stream limits and allow the connection to naturally terminate once remaining streams are consumed. However, the period of time it takes to do so is dependent on the client and an unpredictable closing period might not fit application or operational needs. Applications using QUIC can be conservative with open stream limits in order to reduce the commitment and indeterminism. However, being overly conservative with stream limits affects stream concurrency. Balancing these aspects can be specific to applications and their deployments. Instead of relying on stream limits to avoid abrupt closure, an application-layer graceful close mechanism can be used to communicate the intention to explicitly close the connection at some future point.

HTTP/3 provides such a mechanism using the GOWAWAY frame. In HTTP/3, when the GOAWAY frame is received by a client, it stops opening new streams even if the cumulative stream limit would allow. Instead the client would create a new connection on which to open further streams. Once all streams are closed on the old connection, it can be terminated safely by a connection close or after expiration of the idle time out (see also Section 10).

5. Packetization and Latency

QUIC exposes an interface that provides multiple streams to the application; however, the application usually cannot control how data transmitted over those streams is mapped into frames or how those frames are bundled into packets.

By default, many implementations will try to maximally pack QUIC packets DATA frames from one or more streams to minimize bandwidth consumption and computational costs (see Section 13 of [QUIC]). If there is not enough data available to fill a packet, an implementation might wait for a short time, to optimize bandwidth efficiency instead of latency. This delay can either be pre-configured or dynamically adjusted based on the observed sending pattern of the application.

If the application requires low latency, with only small chunks of data to send, it may be valuable to indicate to QUIC that all data should be send out immediately. Alternatively, if the application expects to use a specific sending pattern, it can also provide a suggested delay to QUIC for how long to wait before bundle frames into a packet.

Similarly, an application has usually no control about the length of a QUIC packet on the wire. QUIC provides the ability to add a PADDING frame to arbitrarily increase the size of packets. Padding

is used by QUIC to ensure that the path is capable of transferring datagrams of at least a certain size, during the handshake (see Sections 8.1 and 14.1 of [QUIC]) and for path validation after connection migration (see Section 8.2 of [QUIC]) as well as for Datagram Packetization Layer PMTU Discovery (DPLMTUD) (see Section 14.3 of [QUIC]).

Padding can also be used by an application to reduce leakage of information about the data that is sent. A QUIC implementation can expose an interface that allows an application layer to specify how to apply padding.

6. Error Handling

QUIC recommends that endpoints signal any detected errors to the peer. Errors can occur at the transport level and the application level. Transport errors, such as a protocol violation, affect the entire connection. Applications that use QUIC can define their own error detection and signaling (see, for example, Section 8 of [QUIC-HTTP]). Application errors can affect an entire connection or a single stream.

QUIC defines an error code space that is used for error handling at the transport layer. QUIC encourages endpoints to use the most specific code, although any applicable code is permitted, including generic ones.

Applications using QUIC define an error code space that is independent from QUIC or other applications (see, for example, Section 8.1 of [QUIC-HTTP]). The values in an application error code space can be reused across connection-level and stream-level errors.

Connection errors lead to connection termination. They are signaled using a CONNECTION_CLOSE frame, which contains an error code and a reason field that can be zero length. Different types of CONNECTION_CLOSE frame are used to signal transport and application errors.

Stream errors lead to stream termination. The are signaled using STOP_SENDING or RESET_STREAM frames, which contain only an error code.

7. ACK-only packets on constrained links

The cost of sending acknowledgments - in processing cost or link utilization - could be a significant proportion of available resources if these resources are constrained. Reducing the rate at which acknowledgments are generated can preserve these resources and improve overall performance, for both network processing as well as application-relevant metrics.

8. Port Selection and Application Endpoint Discovery

In general, port numbers serve two purposes: "first, they provide a demultiplexing identifier to differentiate transport sessions between the same pair of endpoints, and second, they may also identify the application protocol and associated service to which processes connect" [RFC6335]. The assumption that an application can be identified in the network based on the port number is less true today due to encapsulation, mechanisms for dynamic port assignments, and NATs.

As QUIC is a general-purpose transport protocol, there are no requirements that servers use a particular UDP port for QUIC. For applications with a fallback to TCP that do not already have an alternate mapping to UDP, usually the registration (if necessary) and use of the UDP port number corresponding to the TCP port already registered for the application is appropriate. For example, the default port for HTTP/3 [QUIC-HTTP] is UDP port 443, analogous to HTTP/1.1 or HTTP/2 over TLS over TCP.

Additionally, Application-Layer Version Negotiation [RFC7301] permits the client and server to negotiate which of several protocols will be used on a given connection. Therefore, multiple applications might be supported on a single UDP port based on the ALPN token offered. Applications using QUIC should register an ALPN token for use in the TLS handshake.

Applications could define an alternate endpoint discovery mechanism to allow the usage of ports other than the default. For example, HTTP/3 (Sections 3.2 and 3.3 of [QUIC-HTTP]) specifies the use of HTTP Alternative Services for an HTTP origin to advertise the availability of an equivalent HTTP/3 endpoint on a certain UDP port by using the "h3" ALPN token. Note that HTTP/3's ALPN token ("h3") identifies not only the version of the application protocol, but also the version of QUIC itself; this approach allows unambiguous agreement between the endpoints on the protocol stack in use.

Given the prevalence of the assumption in network management practice that a port number maps unambiguously to an application, the use of ports that cannot easily be mapped to a registered service name might lead to blocking or other changes to the forwarding behavior by network elements such as firewalls that use the port number for application identification.

9. Connection Migration

QUIC supports connection migration by the client. If an IP address changes, a QUIC endpoint can still associate packets with an existing transport connection using the destination connection ID field (see also Section 11) in the QUIC header, unless a zero-length value is used. This supports cases where address information changes, such as NAT rebinding, intentional change of the local interface, or based on an indication in the handshake of the server for a preferred address to be used.

Use of a non-zero-length connection ID for the server is strongly recommended if any clients are behind a NAT or could be. A non-zero-length connection ID is also strongly recommended when migration is supported.

Currently QUIC only supports failover cases. Only one "path" can be used at a time; and only when the new path is validated, all traffic can be switched over to that new path. Path validation means that the remote endpoint is required to validate the new path before use in order to avoid address spoofing attacks. Path validation takes at least one RTT and congestion control will also be reset after path migration. Therefore migration usually has a performance impact.

QUIC probing packets, which can be sent on multiple paths at once, are used to perform address validation as well as measure path characteristics as input for the switching decision. Probing packets cannot carry application data but may contain padding frames. Endpoints can use information about their receipt as input to congestion control for that path. Applications could use information learned from probing to inform a decisions to switch paths.

Only the client can actively migrate in version 1 of QUIC. However, servers can indicate during the handshake that they prefer to transfer the connection to a different address after the handshake. For instance, this could be used to move from an address that is shared by multiple servers to an address that is unique to the server instance. The server can provide an IPv4 and an IPv6 address in a transport parameter during the TLS handshake and the client can select between the two if both are provided. See also Section 9.6 of [QUIC].

10. Connection Termination

QUIC connections are terminated in one of three ways: implicit idle timeout, explicit immediate close, or explicit stateless reset.

QUIC does not provide any mechanism for graceful connection termination; applications using QUIC can define their own graceful termination process (see, for example, Section 5.2 of [QUIC-HTTP]).

QUIC idle timeout is enabled via transport parameters. Client and server announce a timeout period and the effective value for the connection is the minimum of the two values. After the timeout period elapses, the connection is silently closed. An application therefore should be able to configure its own maximum value, as well as have access to the computed minimum value for this connection. An application may adjust the maximum idle timeout for new connections based on the number of open or expected connections, since shorter timeout values may free-up resources more quickly.

Application data exchanged on streams or in datagrams defers the QUIC idle timeout. Applications that provide their own keep-alive mechanisms will therefore keep a QUIC connection alive. Applications that do not provide their own keep-alive can use transport-layer mechanisms (see Section 10.1.2 of [QUIC], and Section 3.2). However, QUIC implementation interfaces for controlling such transport behavior can vary, affecting the robustness of such approaches.

An immediate close is signaled by a CONNECTION_CLOSE frame (see Section 6). Immediate close causes all streams to become immediately closed, which may affect applications; see Section 4.5.

A stateless reset is an option of last resort for an endpoint that does not have access to connection state. Receiving a stateless reset is an indication of an unrecoverable error distinct from connection errors in that there is no application-layer information provided.

11. Information Exposure and the Connection ID

QUIC exposes some information to the network in the unencrypted part of the header, either before the encryption context is established or because the information is intended to be used by the network. QUIC has a long header that exposes some additional information (the version and the source connection ID), while the short header exposes only the destination connection ID. In QUIC version 1, the long header is used during connection establishment, while the short header is used for data transmission in an established connection.

The connection ID can be zero length. Zero length connection IDs can be chosen on each endpoint individually, on any packet except the first packets sent by clients during connection establishment.

An endpoint that selects a zero-length connection ID will receive packets with a zero-length destination connection ID. The endpoint needs to use other information, such as the source and destination IP address and port number to identify which connection is referred to. This could mean that the endpoint is unable to match datagrams to connections successfully if these values change, making the connection effectively unable to survive NAT rebinding or migrate to a new path.

11.1. Server-Generated Connection ID

QUIC supports a server-generated connection ID, transmitted to the client during connection establishment (see Section 7.2 of [QUIC]). Servers behind load balancers may need to change the connection ID during the handshake, encoding the identity of the server or information about its load balancing pool, in order to support stateless load balancing.

Server deployments with load balancers and other routing infrastructure need to ensure that this infrastructure consistently routes packets to the server instance that has the connection state, even if addresses, ports, and/or connection IDs change. This might require coordination between servers and infrastructure. One method of achieving this involves encoding routing information into the connection ID. For an example of this technique, see [QUIC-LB].

11.2. Mitigating Timing Linkability with Connection ID Migration

QUIC requires that endpoints generate fresh connection IDs for use on new network paths. Choosing values that are unlinkable to an outside observer ensures that activity on different paths cannot be trivially correlated using the connection ID.

While sufficiently robust connection ID generation schemes will mitigate linkability issues, they do not provide full protection. Analysis of the lifetimes of six-tuples (source and destination addresses as well as the migrated CID) may expose these links anyway.

In the limit where connection migration in a server pool is rare, it is trivial for an observer to associate two connection IDs. Conversely, in the opposite limit where every server handles multiple simultaneous migrations, even an exposed server mapping may be insufficient information.

The most efficient mitigations for these attacks are through network design and/or operational practice, by using a load balancing architecture that loads more flows onto a single server-side address, by coordinating the timing of migrations in an attempt to increase the number of simultaneous migrations at a given time, or through other means.

11.3. Using Server Retry for Redirection

QUIC provides a Server Retry packet that can be sent by a server in response to the Client Initial packet. The server may choose a new connection ID in that packet and the client will retry by sending another Client Initial packet with the server-selected connection ID. This mechanism can be used to redirect a connection to a different server, e.g. due to performance reasons or when servers in a server pool are upgraded gradually, and therefore may support different versions of QUIC. In this case, it is assumed that all servers belonging to a certain pool are served in cooperation with load balancers that forward the traffic based on the connection ID. A server can choose the connection ID in the Server Retry packet such that the load balancer will redirect the next Client Initial packet to a different server in that pool. Alternatively the load balancer can directly offer a Retry service as further described in [QUIC-LB].

Section 4 of [RFC5077] describes an example approach for constructing TLS resumption tickets that can be also applied for validation tokens, however, the use of more modern cryptographic algorithms is highly recommended.

12. Quality of Service (QoS) and DSCP

QUIC assumes that all packets of a QUIC connection, or at least with the same 5-tuple {dest addr, source addr, protocol, dest port, source port}, will receive similar network treatment since feedback about loss or delay of each packet is used as input to the congestion controller. Therefore it is not recommended to use different DiffServ Code Points (DSCPs) [RFC2475] for packets belonging to the same connection. If differential network treatment, e.g. by the use of different DSCPs, is desired, multiple QUIC connections to the same server may be used. However, in general it is recommended to minimize the number of QUIC connections to the same server, to avoid increased overheads and, more importantly, competing congestion control.

13. Use of Versions and Cryptographic Handshake

Versioning in QUIC may change the protocol's behavior completely, except for the meaning of a few header fields that have been declared to be invariant [QUIC-INVARIANTS]. A version of QUIC with a higher version number will not necessarily provide a better service, but might simply provide a different feature set. As such, an application needs to be able to select which versions of QUIC it wants to use.

A new version could use an encryption scheme other than TLS 1.3 or higher. [QUIC] specifies requirements for the cryptographic handshake as currently realized by TLS 1.3 and described in a separate specification [QUIC-TLS]. This split is performed to enable light-weight versioning with different cryptographic handshakes.

14. Enabling New Versions

QUIC version 1 does not specify a version negotation mechanism in the base spec but [I-D.draft-ietf-quic-version-negotiation] proposes an extension. This process assumes that the set of versions that a server supports is fixed. This complicates the process for deploying new QUIC versions or disabling old versions when servers operate in clusters.

A server that rolls out a new version of QUIC can do so in three stages. Each stage is completed across all server instances before moving to the next stage.

In the first stage of deployment, all server instances start accepting new connections with the new version. The new version can be enabled progressively across a deployment, which allows for selective testing. This is especially useful when the new version is compatible with an old version, because the new version is more likely to be used.

While enabling the new version, servers do not advertise the new version in any Version Negotiation packets they send. This prevents clients that receive a Version Negotiation packet from attempting to connect to server instances that might not have the new version enabled.

During the initial deployment, some clients will have received Version Negotiation packets that indicate that the server does not support the new version. Other clients might have successfully connected with the new version and so will believe that the server supports the new version. Therefore, servers need to allow for this ambiguity when validating the negotiated version.

The second stage of deployment commences once all server instances are able accept new connections with the new version. At this point, all servers can start sending the new version in Version Negotiation packets.

During the second stage, the server still allows for the possibility that some clients believe the new version to be available and some do not. This state will persist only for as long as any Version Negotiation packets take to be transmitted and responded to. So the third stage can follow after a relatively short delay.

The third stage completes the process by enabling authentication of the negotiated version with the assumption that the new version is fully available.

The process for disabling an old version or rolling back the introduction of a new version uses the same process in reverse. Servers disable validation of the old version, stop sending the old version in Version Negotiation packets, then the old version is no longer accepted.

15. Unreliable Datagram Service over QUIC

[I-D.ietf-quic-datagram] specifies a QUIC extension to enable sending and receiving unreliable datagrams over QUIC. Unlike operating directly over UDP, applications that use the QUIC datagram service do not need to implement their own congestion control, per [RFC8085], as QUIC datagrams are congestion controlled.

QUIC datagrams are not flow-controlled, and as such data chunks may be dropped if the receiver is overloaded. While the reliable transmission service of QUIC provides a stream-based interface to send and receive data in order over multiple QUIC streams, the datagram service has a unordered message-based interface. If needed, an application layer framing can be used on top to allow separate flows of unreliable datagrams to be multiplexed on one QUIC connection.

16. IANA Considerations

This document has no actions for IANA; however, note that Section 8 recommends that application bindings to QUIC for applications using TCP register UDP ports analogous to their existing TCP registrations.

17. Security Considerations

See the security considerations in [QUIC] and [QUIC-TLS]; the security considerations for the underlying transport protocol are relevant for applications using QUIC, as well. Considerations on linkability, replay attacks, and randomness discussed in [QUIC-TLS] should be taken into account when deploying and using QUIC.

Application developers should note that any fallback they use when QUIC cannot be used due to network blocking of UDP should guarantee the same security properties as QUIC; if this is not possible, the connection should fail to allow the application to explicitly handle fallback to a less-secure alternative. See Section 2.

Further, [QUIC-HTTP] provides security considerations specific to HTTP. However, discussions such as on cross-protocol attacks, traffic analysis and padding, or migration might be relevant for other applications using QUIC as well.

18. Contributors

The following people have contributed text to this document:

* Igor Lubashev
* Mike Bishop
* Martin Thomson
* Lucas Pardue
* Gorry Fairhurst

19. Acknowledgments

Thanks also to Martin Duke, Sean Turner, and Ian Swett for their reviews.

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.

20. References

20.1. Normative References

[QUIC] Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", Work in Progress, Internet-Draft, draft-ietf-quic-transport-34, 14 January 2021, <https://tools.ietf.org/html/draft-ietf-quic-transport-34>.

[QUIC-INVARIANTS] Thomson, M., "Version-Independent Properties of QUIC", Work in Progress, Internet-Draft, draft-ietf-quic-invariants-13, 14 January 2021, <https://tools.ietf.org/html/draft-ietf-quic-invariants-13>.

[QUIC-TLS] Thomson, M. and S. Turner, "Using TLS to Secure QUIC", Work in Progress, Internet-Draft, draft-ietf-quic-tls-34, 14 January 2021, <https://tools.ietf.org/html/draft-ietf-quic-tls-34>.

[RFC6335] Cotton, M., Eggert, L., Touch, J., Westerlund, M., and S. Cheshire, "Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry", BCP 165, RFC 6335, DOI 10.17487/RFC6335, August 2011, <https://www.rfc-editor.org/rfc/rfc6335>.

[TLS13] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/rfc/rfc8446>.

20.2. Informative References

[Edeline16] Edeline, K., Kuehlewind, M., Trammell, B., Aben, E., and B. Donnet, "Using UDP for Internet Transport Evolution (arXiv preprint 1612.07816)", 22 December 2016, <https://arxiv.org/abs/1612.07816>.

[Hatonen10] 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.

[HTTP-REPLAY] Thomson, M., Nottingham, M., and W. Tarreau, "Using Early Data in HTTP", RFC 8470, DOI 10.17487/RFC8470, September 2018, <https://www.rfc-editor.org/rfc/rfc8470>.

[I-D.draft-ietf-httpbis-priority]
Oku, K. and L. Pardue, "Extensible Prioritization Scheme for HTTP", Work in Progress, Internet-Draft, draft-ietf-httpbis-priority-03, 11 January 2021, <https://tools.ietf.org/html/draft-ietf-httpbis-priority-03>.

[I-D.draft-ietf-quic-version-negotiation]
Schinazi, D. and E. Rescorla, "Compatible Version Negotiation for QUIC", Work in Progress, Internet-Draft, draft-ietf-quic-version-negotiation-03, 4 February 2021, <https://tools.ietf.org/html/draft-ietf-quic-version-negotiation-03>.

[I-D.ietf-quic-datagram]
Pauly, T., Kinnear, E., and D. Schinazi, "An Unreliable Datagram Extension to QUIC", Work in Progress, Internet-Draft, draft-ietf-quic-datagram-02, 16 February 2021, <https://tools.ietf.org/html/draft-ietf-quic-datagram-02>.

[I-D.ietf-taps-arch]
Pauly, T., Trammell, B., Brunstrom, A., Fairhurst, G., Perkins, C., Tiesel, P. S., and C. A. Wood, "An Architecture for Transport Services", Work in Progress, Internet-Draft, draft-ietf-taps-arch-09, 2 November 2020, <https://tools.ietf.org/html/draft-ietf-taps-arch-09>.

[PaaschNanog]
Paasch, C., "Network Support for TCP Fast Open (NANOG 67 presentation)", 13 June 2016, <https://www.nanog.org/sites/default/files/Paasch_Network_Support.pdf>.

[QUIC-HTTP]
Bishop, M., "Hypertext Transfer Protocol Version 3 (HTTP/3)", Work in Progress, Internet-Draft, draft-ietf-quic-http-34, 2 February 2021, <https://tools.ietf.org/html/draft-ietf-quic-http-34>.

[QUIC-LB] Duke, M. and N. Banks, "QUIC-LB: Generating Routable QUIC Connection IDs", Work in Progress, Internet-Draft, draft-ietf-quic-load-balancers-06, 4 February 2021, <https://tools.ietf.org/html/draft-ietf-quic-load-balancers-06>.

[RFC2475] Blake, S., Black, D., Carlson, M., Davies, E., Wang, Z., and W. Weiss, "An Architecture for Differentiated Services", RFC 2475, DOI 10.17487/RFC2475, December 1998, <https://www.rfc-editor.org/rfc/rfc2475>.

[RFC5077] Salowey, J., Zhou, H., Eronen, P., and H. Tschofenig, "Transport Layer Security (TLS) Session Resumption without Server-Side State", RFC 5077, DOI 10.17487/RFC5077, January 2008, <https://www.rfc-editor.org/rfc/rfc5077>.

[RFC5382] Guha, S., Ed., Biswas, K., Ford, B., Sivakumar, S., and P. Srisuresh, "NAT Behavioral Requirements for TCP", BCP 142, RFC 5382, DOI 10.17487/RFC5382, October 2008, <https://www.rfc-editor.org/rfc/rfc5382>.

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

[RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <https://www.rfc-editor.org/rfc/rfc7413>.

[RFC8085] Eggert, L., Fairhurst, G., and G. Shepherd, "UDP Usage Guidelines", BCP 145, RFC 8085, DOI 10.17487/RFC8085, March 2017, <https://www.rfc-editor.org/rfc/rfc8085>.

[Swett16] Swett, I., "QUIC Deployment Experience at Google (IETF96 QUIC BoF presentation)", 20 July 2016, <https://www.ietf.org/proceedings/96/slides/slides-96-quic-3.pdf>.

[Trammell16] Trammell, B. and M. Kuehlewind, "Internet Path Transparency Measurements using RIPE Atlas (RIPE72 MAT presentation)", 25 May 2016, <https://ripe72.ripe.net/wp-content/uploads/presentations/86-atlas-udpdiff.pdf>.

Authors' Addresses

Mirja Kuehlewind
Ericsson

Email: mirja.kuehlewind@ericsson.com

Brian Trammell
Google
Gustav-Gull-Platz 1
CH- 8004 Zurich
Switzerland

Email: ietf@trammell.ch

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: 20 August 2021

T. Pauly
E. Kinnear
Apple Inc.
D. Schinazi
Google LLC

16 February 2021

An Unreliable Datagram Extension to QUIC
draft-ietf-quic-datagram-02

Abstract

This document defines an extension to the QUIC transport protocol to add support for sending and receiving unreliable datagrams over a QUIC connection.

Discussion of this work is encouraged to happen on the QUIC IETF mailing list quic@ietf.org (mailto:quic@ietf.org) or on the GitHub repository which contains the draft: https://github.com/quicwg/datagram (https://github.com/quicwg/datagram).

Table of Contents

1. Introduction

The QUIC Transport Protocol [I-D.ietf-quic-transport] provides a secure, multiplexed connection for transmitting reliable streams of application data. Reliability within QUIC is performed on a per-stream basis, so some frame types are not eligible for retransmission.

Some applications, particularly those that need to transmit real-time data, prefer to transmit data unreliably. These applications can build directly upon UDP [RFC0768] as a transport, and can add security with DTLS [RFC6347]. Extending QUIC to support transmitting unreliable application data would provide another option for secure datagrams, with the added benefit of sharing a cryptographic and authentication context used for reliable streams.

This document defines two new DATAGRAM QUIC frame types, which carry application data without requiring retransmissions.

Discussion of this work is encouraged to happen on the QUIC IETF mailing list quic@ietf.org (mailto:quic@ietf.org) or on the GitHub repository which contains the draft: https://github.com/quicwg/datagram (https://github.com/quicwg/datagram).

1.1. Specification of Requirements

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

2. Motivation

Transmitting unreliable data over QUIC provides benefits over existing solutions:

* Applications that open both a reliable TLS stream and an unreliable DTLS flow to the same peer can benefit by sharing a single handshake and authentication context between a reliable QUIC stream and flow of unreliable QUIC datagrams. This can reduce the latency required for handshakes.

* QUIC uses a more nuanced loss recovery mechanism than the DTLS handshake, which has a basic packet loss retransmission timer. This may allow loss recovery to occur more quickly for QUIC data.

* QUIC datagrams, while unreliable, can support acknowledgements, allowing applications to be aware of whether a datagram was successfully received.

* QUIC datagrams are subject to QUIC congestion control, allowing applications to avoid implementing their own.

These reductions in connection latency, and application insight into the delivery of datagrams, can be useful for optimizing audio/video streaming applications, gaming applications, and other real-time network applications.

Unreliable QUIC datagrams can also be used to implement an IP packet tunnel over QUIC, such as for a Virtual Private Network (VPN). Internet-layer tunneling protocols generally require a reliable and authenticated handshake, followed by unreliable secure transmission of IP packets. This can, for example, require a TLS connection for the control data, and DTLS for tunneling IP packets. A single QUIC connection could support both parts with the use of unreliable datagrams.

3. Transport Parameter

Support for receiving the DATAGRAM frame types is advertised by means of a QUIC Transport Parameter (name=max_datagram_frame_size, value=0x0020). The max_datagram_frame_size transport parameter is an integer value (represented as a variable-length integer) that represents the maximum size of a DATAGRAM frame (including the frame type, length, and payload) the endpoint is willing to receive, in bytes. An endpoint that includes this parameter supports the DATAGRAM frame types and is willing to receive such frames on this connection. Endpoints MUST NOT send DATAGRAM frames until they have received the max_datagram_frame_size transport parameter. Endpoints MUST NOT send DATAGRAM frames of size strictly larger than the value of max_datagram_frame_size the endpoint has received from its peer. An endpoint that receives a DATAGRAM frame when it has not sent the max_datagram_frame_size transport parameter MUST terminate the connection with error PROTOCOL_VIOLATION. An endpoint that receives a DATAGRAM frame that is strictly larger than the value it sent in its max_datagram_frame_size transport parameter MUST terminate the connection with error PROTOCOL_VIOLATION. Endpoints that wish to use DATAGRAM frames need to ensure they send a max_datagram_frame_size value sufficient to allow their peer to use them. It is RECOMMENDED to send the value 65535 in the max_datagram_frame_size transport parameter as that indicates to the peer that this endpoint will accept any DATAGRAM frame that fits inside a QUIC packet.

The max_datagram_frame_size transport parameter is a unidirectional limit and indication of support of DATAGRAM frames. Application protocols that use DATAGRAM frames MAY choose to only negotiate and use them in a single direction.

When clients use 0-RTT, they MAY store the value of the server's max_datagram_frame_size transport parameter. Doing so allows the client to send DATAGRAM frames in 0-RTT packets. When servers decide to accept 0-RTT data, they MUST send a max_datagram_frame_size transport parameter greater or equal to the value they sent to the client in the connection where they sent them the NewSessionTicket message. If a client stores the value of the max_datagram_frame_size transport parameter with their 0-RTT state, they MUST validate that the new value of the max_datagram_frame_size transport parameter sent by the server in the handshake is greater or equal to the stored value; if not, the client MUST terminate the connection with error PROTOCOL_VIOLATION.

Application protocols that use datagrams MUST define how they react to the max_datagram_frame_size transport parameter being missing. If datagram support is integral to the application, the application protocol can fail the handshake if the max_datagram_frame_size transport parameter is not present.

4. Datagram Frame Type

DATAGRAM frames are used to transmit application data in an unreliable manner. The DATAGRAM frame type takes the form 0b0011000X (or the values 0x30 and 0x31). The least significant bit of the DATAGRAM frame type is the LEN bit (0x01). It indicates that there is a Length field present. If this bit is set to 0, the Length field is absent and the Datagram Data field extends to the end of the packet. If this bit is set to 1, the Length field is present.

The DATAGRAM frame is structured as follows:

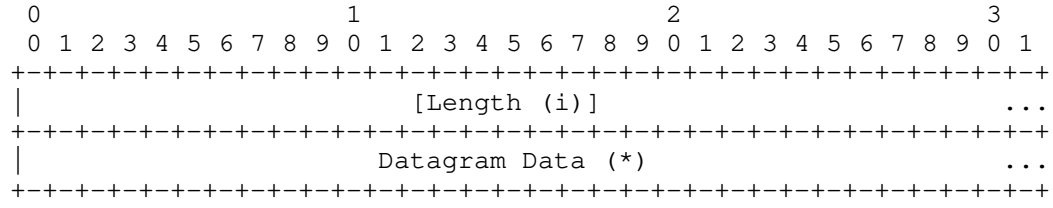


Figure 1: DATAGRAM Frame Format

DATAGRAM frames contain the following fields:

Length: A variable-length integer specifying the length of the datagram in bytes. This field is present only when the LEN bit is set. If the LEN bit is not set, the datagram data extends to the end of the QUIC packet. Note that empty (i.e., zero-length) datagrams are allowed.

Datagram Data: The bytes of the datagram to be delivered.

5. Behavior and Usage

When an application sends an unreliable datagram over a QUIC connection, QUIC will generate a new DATAGRAM frame and send it in the first available packet. This frame SHOULD be sent as soon as possible, and MAY be coalesced with other frames.

When a QUIC endpoint receives a valid DATAGRAM frame, it SHOULD deliver the data to the application immediately, as long as it is able to process the frame and can store the contents in memory.

DATAGRAM frames MUST be protected with either 0-RTT or 1-RTT keys.

Application protocols using datagrams are responsible for defining the semantics of the Datagram Data field, and how it is parsed. If the application protocol supports the coexistence of multiple entities using datagrams inside a single QUIC connection, it may need a mechanism to allow demultiplexing between them. For example, using datagrams with HTTP/3 involves prepending a flow identifier to all datagrams, see [I-D.schinazi-quic-h3-datagram].

Note that while the max_datagram_frame_size transport parameter places a limit on the maximum size of DATAGRAM frames, that limit can be further reduced by the max_packet_size transport parameter, and by the Maximum Transmission Unit (MTU) of the path between endpoints. DATAGRAM frames cannot be fragmented, therefore application protocols need to handle cases where the maximum datagram size is limited by other factors.

5.1. Acknowledgement Handling

Although DATAGRAM frames are not retransmitted upon loss detection, they are ack-eliciting ([I-D.ietf-quic-recovery]). Receivers SHOULD support delaying ACK frames (within the limits specified by max_ack_delay) in reponse to receiving packets that only contain DATAGRAM frames, since the timing of these acknowledgements is not used for loss recovery.

If a sender detects that a packet containing a specific DATAGRAM frame might have been lost, the implementation MAY notify the application that it believes the datagram was lost. Similarly, if a packet containing a DATAGRAM frame is acknowledged, the implementation MAY notify the application that the datagram was successfully transmitted and received. Note that, due to reordering, a DATAGRAM frame that was thought to be lost could at a later point be received and acknowledged.

5.2. Flow Control

DATAGRAM frames do not provide any explicit flow control signaling, and do not contribute to any per-flow or connection-wide data limit.

The risk associated with not providing flow control for DATAGRAM frames is that a receiver may not be able to commit the necessary resources to process the frames. For example, it may not be able to store the frame contents in memory. However, since DATAGRAM frames are inherently unreliable, they MAY be dropped by the receiver if the receiver cannot process them.

5.3. Congestion Control

DATAGRAM frames employ the QUIC connection's congestion controller. As a result, a connection may be unable to send a DATAGRAM frame generated by the application until the congestion controller allows it [I-D.ietf-quic-recovery]. The sender implementation MUST either delay sending the frame until the controller allows it or drop the frame without sending it (at which point it MAY notify the application).

Implementations can optionally support allowing the application to specify a sending expiration time, beyond which a congestion-controlled DATAGRAM frame ought to be dropped without transmission.

6. Security Considerations

The DATAGRAM frame shares the same security properties as the rest of the data transmitted within a QUIC connection. All application data transmitted with the DATAGRAM frame, like the STREAM frame, MUST be protected either by 0-RTT or 1-RTT keys.

7. IANA Considerations

This document registers a new value in the QUIC Transport Parameter Registry:

Value: 0x0020 (if this document is approved)

Parameter Name: max_datagram_frame_size

Specification: Indicates that the connection should enable support for unreliable DATAGRAM frames. An endpoint that advertises this transport parameter can receive datagrams frames from the other endpoint, up to and including the length in bytes provided in the transport parameter.

This document also registers a new value in the QUIC Frame Type registry:

Value: 0x30 and 0x31 (if this document is approved)

Frame Name: DATAGRAM

Specification: Unreliable application data

8. Acknowledgments

Thanks to Ian Swett, who inspired this proposal.

9. References

9.1. Normative References

[I-D.ietf-quic-recovery]
Iyengar, J. and I. Swett, "QUIC Loss Detection and Congestion Control", Work in Progress, Internet-Draft, draft-ietf-quic-recovery-34, 14 January 2021, <http://www.ietf.org/internet-drafts/draft-ietf-quic-recovery-34.txt>.

[I-D.ietf-quic-transport]
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", Work in Progress, Internet-Draft, draft-ietf-quic-transport-34, 14 January 2021, <http://www.ietf.org/internet-drafts/draft-ietf-quic-transport-34.txt>.

9.2. Informative References

[I-D.schinazi-quic-h3-datagram]
Schinazi, D., "Using QUIC Datagrams with HTTP/3", Work in Progress, Internet-Draft, draft-schinazi-quic-h3-datagram-05, 12 October 2020, <http://www.ietf.org/internet-drafts/draft-schinazi-quic-h3-datagram-05.txt>.

[RFC0768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, DOI 10.17487/RFC0768, August 1980, <https://www.rfc-editor.org/info/rfc768>.

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

[RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <https://www.rfc-editor.org/info/rfc6347>.

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

Authors' Addresses

Tommy Pauly
Apple Inc.
One Apple Park Way
Cupertino, California 95014,
United States of America

Email: tpauly@apple.com

Eric Kinnear
Apple Inc.
One Apple Park Way
Cupertino, California 95014,
United States of America

Email: ekinnear@apple.com

David Schinazi
Google LLC
1600 Amphitheatre Parkway
Mountain View, California 94043,
United States of America

Email: dschinazi.ietf@gmail.com

QUIC
Internet-Draft
Intended status: Standards Track
Expires: 8 August 2021

M. Duke
F5 Networks, Inc.
N. Banks
Microsoft

4 February 2021

QUIC-LB: Generating Routable QUIC Connection IDs
draft-ietf-quic-load-balancers-06

Abstract

The QUIC protocol design is resistant to transparent packet inspection, injection, and modification by intermediaries. However, the server can explicitly cooperate with network services by agreeing to certain conventions and/or sharing state with those services. This specification provides a standardized means of solving three problems: (1) maintaining routability to servers via a low-state load balancer even when the connection IDs in use change; (2) explicit encoding of the connection ID length in all packets to assist hardware accelerators; and (3) injection of QUIC Retry packets by an anti-Denial-of-Service agent on behalf of the server.

Status of This Memo

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

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

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

This Internet-Draft will expire on 8 August 2021.

Table of Contents

1. Introduction

QUIC packets [QUIC-TRANSPORT] usually contain a connection ID to allow endpoints to associate packets with different address/ port 4-tuples to the same connection context. This feature makes connections robust in the event of NAT rebinding. QUIC endpoints usually designate the connection ID which peers use to address packets. Server-generated connection IDs create a potential need for out-of-band communication to support QUIC.

QUIC allows servers (or load balancers) to designate an initial connection ID to encode useful routing information for load balancers. It also encourages servers, in packets protected by cryptography, to provide additional connection IDs to the client. This allows clients that know they are going to change IP address or port to use a separate connection ID on the new path, thus reducing linkability as clients move through the world.

There is a tension between the requirements to provide routing information and mitigate linkability. Ultimately, because new connection IDs are in protected packets, they must be generated at the server if the load balancer does not have access to the connection keys. However, it is the load balancer that has the context necessary to generate a connection ID that encodes useful routing information. In the absence of any shared state between load balancer and server, the load balancer must maintain a relatively expensive table of server-generated connection IDs, and will not route packets correctly if they use a connection ID that was originally communicated in a protected NEW_CONNECTION_ID frame.

This specification provides common algorithms for encoding the server mapping in a connection ID given some shared parameters. The mapping is generally only discoverable by observers that have the parameters, preserving unlinkability as much as possible.

Aside from load balancing, a QUIC server may also desire to offload other protocol functions to trusted intermediaries. These intermediaries might include hardware assist on the server host itself, without access to fully decrypted QUIC packets. For example, this document specifies a means of offloading stateless retry to counter Denial of Service attacks. It also proposes a system for self-encoding connection ID length in all packets, so that crypto offload can consistently look up key information.

While this document describes a small set of configuration parameters to make the server mapping intelligible, the means of distributing these parameters between load balancers, servers, and other trusted intermediaries is out of its scope. There are numerous well-known infrastructures for distribution of configuration.

1.1. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

In this document, these words will appear with that interpretation only when in ALL CAPS. Lower case uses of these words are not to be interpreted as carrying significance described in RFC 2119.

In this document, "client" and "server" refer to the endpoints of a QUIC connection unless otherwise indicated. A "load balancer" is an intermediary for that connection that does not possess QUIC connection keys, but it may rewrite IP addresses or conduct other IP or UDP processing. A "configuration agent" is the entity that determines the QUIC-LB configuration parameters for the network and leverages some system to distribute that configuration.

Note that stateful load balancers that act as proxies, by terminating a QUIC connection with the client and then retrieving data from the server using QUIC or another protocol, are treated as a server with respect to this specification.

For brevity, "Connection ID" will often be abbreviated as "CID".

1.2. Notation

All wire formats will be depicted using the notation defined in Section 1.3 of [QUIC-TRANSPORT]. There is one addition: the function len() refers to the length of a field which can serve as a limit on a different field, so that the lengths of two fields can be concisely defined as limited to a sum, for example:

x(A..B) y(C..B-len(x))

indicates that x can be of any length between A and B, and y can be of any length between C and B provided that (len(x) + len(y)) does not exceed B.

The example below illustrates the basic framework:

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

Figure 1: Example Format

2. Protocol Objectives

2.1. Simplicity

QUIC is intended to provide unlinkability across connection migration, but servers are not required to provide additional connection IDs that effectively prevent linkability. If the coordination scheme is too difficult to implement, servers behind load balancers using connection IDs for routing will use trivially linkable connection IDs. Clients will therefore be forced to choose between terminating the connection during migration or remaining linkable, subverting a design objective of QUIC.

The solution should be both simple to implement and require little additional infrastructure for cryptographic keys, etc.

2.2. Security

In the limit where there are very few connections to a pool of servers, no scheme can prevent the linking of two connection IDs with high probability. In the opposite limit, where all servers have many connections that start and end frequently, it will be difficult to associate two connection IDs even if they are known to map to the same server.

QUIC-LB is relevant in the region between these extremes: when the information that two connection IDs map to the same server is helpful to linking two connection IDs. Obviously, any scheme that transparently communicates this mapping to outside observers compromises QUIC's defenses against linkability.

Though not an explicit goal of the QUIC-LB design, concealing the server mapping also complicates attempts to focus attacks on a specific server in the pool.

3. First CID octet

The first octet of a Connection ID is reserved for two special purposes, one mandatory (config rotation) and one optional (length self-description).

Subsequent sections of this document refer to the contents of this octet as the "first octet."

3.1. Config Rotation

The first two bits of any connection ID MUST encode an identifier for the configuration that the connection ID uses. This enables incremental deployment of new QUIC-LB settings (e.g., keys).

When new configuration is distributed to servers, there will be a transition period when connection IDs reflecting old and new configuration coexist in the network. The rotation bits allow load balancers to apply the correct routing algorithm and parameters to incoming packets.

Configuration Agents SHOULD deliver new configurations to load balancers before doing so to servers, so that load balancers are ready to process CIDs using the new parameters when they arrive.

A Configuration Agent SHOULD NOT use a codepoint to represent a new configuration until it takes precautions to make sure that all connections using CIDs with an old configuration at that codepoint have closed or transitioned.

Servers MUST NOT generate new connection IDs using an old configuration after receiving a new one from the configuration agent. Servers MUST send NEW_CONNECTION_ID frames that provide CIDs using the new configuration, and retire CIDs using the old configuration using the "Retire Prior To" field of that frame.

It also possible to use these bits for more long-lived distinction of different configurations, but this has privacy implications (see Section 11.3).

3.2. Configuration Failover

If a server has not received a valid QUIC-LB configuration, and believes that low-state, Connection-ID aware load balancers are in the path, it SHOULD generate connection IDs with the config rotation bits set to '11' and SHOULD use the "disable_active_migration" transport parameter in all new QUIC connections. It SHOULD NOT send NEW_CONNECTION_ID frames with new values.

A load balancer that sees a connection ID with config rotation bits set to '11' MUST revert to 5-tuple routing.

3.3. Length Self-Description

Local hardware cryptographic offload devices may accelerate QUIC servers by receiving keys from the QUIC implementation indexed to the connection ID. However, on physical devices operating multiple QUIC servers, it is impractical to efficiently lookup these keys if the connection ID does not self-encode its own length.

Note that this is a function of particular server devices and is irrelevant to load balancers. As such, load balancers MAY omit this from their configuration. However, the remaining 6 bits in the first octet of the Connection ID are reserved to express the length of the following connection ID, not including the first octet.

A server not using this functionality SHOULD make the six bits appear to be random.

3.4. Format

```
First Octet {
  Config Rotation (2),
  CID Len or Random Bits (6),
}
```

Figure 2: First Octet Format

The first octet has the following fields:

Config Rotation: Indicates the configuration used to interpret the CID.

CID Len or Random Bits: Length Self-Description (if applicable), or random bits otherwise. Encodes the length of the Connection ID following the First Octet.

4. Load Balancing Preliminaries

In QUIC-LB, load balancers do not generate individual connection IDs for servers. Instead, they communicate the parameters of an algorithm to generate routable connection IDs.

The algorithms differ in the complexity of configuration at both load balancer and server. Increasing complexity improves obfuscation of the server mapping.

This section describes three participants: the configuration agent, the load balancer, and the server. For any given QUIC-LB configuration that enables connection-ID-aware load balancing, there must be a choice of (1) routing algorithm, (2) server ID allocation strategy, and (3) algorithm parameters.

Fundamentally, servers generate connection IDs that encode their server ID. Load balancers decode the server ID from the CID in incoming packets to route to the correct server.

There are situations where a server pool might be operating two or more routing algorithms or parameter sets simultaneously. The load balancer uses the first two bits of the connection ID to multiplex incoming DCIDs over these schemes (see Section 3.1).

4.1. Non-Compliant Connection IDs

QUIC-LB servers will generate Connection IDs that are decodable to extract a server ID in accordance with a specified algorithm and parameters. However, QUIC often uses client-generated Connection IDs prior to receiving a packet from the server.

These client-generated CIDs might not conform to the expectations of the routing algorithm and therefore not be routable by the load balancer. These are called "non-compliant DCIDs":

* The config rotation bits (Section 3.1) may not correspond to an active configuration. Note: a packet with a DCID that indicates 5-tuple routing (see Section 3.2) is always compliant.

* The DCID might not be long enough for the decoder to process.

* The extracted server mapping might not correspond to an active server.

All other DCIDs are compliant.

Load balancers MUST forward packets with compliant DCIDs to a server in accordance with the chosen routing algorithm.

Load balancers SHOULD drop packets with non-compliant DCIDs in a short header.

The routing of long headers with non-compliant DCIDs depends on the server ID allocation strategy, described in Section 4.3. However, the load balancer MUST NOT drop these packets, with one exception.

Load balancers MAY drop packets with long headers and non-compliant DCIDs if and only if it knows that the encoded QUIC version does not allow a non- compliant DCID in a packet with that signature. For example, a load balancer can safely drop a QUIC version 1 Handshake packet with a non-compliant DCID, as a version 1 Handshake packet sent to a QUIC-LB compliant server will always have a server-generated compliant CID. The prohibition against dropping packets with long headers remains for unknown QUIC versions.

Furthermore, while the load balancer function MUST NOT drop packets, the device might implement other security policies, outside the scope of this specification, that might force a drop.

Servers that receive packets with noncompliant CIDs MUST use the available mechanisms to induce the client to use a compliant CID in future packets. In QUIC version 1, this requires using a compliant CID in the Source CID field of server-generated long headers.

4.2. Arbitrary Algorithms

There are conditions described below where a load balancer routes a packet using an "arbitrary algorithm." It can choose any algorithm, without coordination with the servers, but the algorithm SHOULD be deterministic over short time scales so that related packets go to the same server. The design of this algorithm SHOULD consider the version-invariant properties of QUIC described in [QUIC-INVARIANTS] to maximize its robustness to future versions of QUIC.

An arbitrary algorithmr MUST NOT make the routing behavior dependent on any bits in the first octet of the QUIC packet header, except the first bit, which indicates a long header. All other bits are QUIC version-dependent and intermediaries should not base their design on version-specific templates.

For example, one arbitrary algorithm might convert a non-compliant DCID to an integer and divided by the number of servers, with the modulus used to forward the packet. The number of servers is usually consistent on the time scale of a QUIC connection handshake. Another might simply hash the address/port 4-tuple. See also Section 10.

4.3. Server ID Allocation

For any given configuration, the configuration agent must specify if server IDs will be statically or dynamically allocated. Load Balancer configurations with statically allocated server IDs explicitly include a mapping of server IDs to forwarding addresses. The corresponding server configurations contain one or more unique server IDs.

A dynamically allocated configuration does not include any bespoke assignment, reducing configuration complexity. However, it places limits on the maximum server ID length and requires more state at the load balancer. In certain edge cases, it can force parts of the system to fail over to 5-tuple routing for a short time.

In either case, the configuration agent chooses a server ID length for each configuration that MUST be at least one octet. For Static Allocation, the maximum length depends on the algorithm. For dynamic allocation, the maximum length is 7 octets.

A QUIC-LB configuration MAY significantly over-provision the server ID space (i.e., provide far more codepoints than there are servers) to increase the probability that a randomly generated Destination Connection ID is non- compliant.

Conceptually, each configuration has its own set of server ID allocations, though two static configurations with identical server ID lengths MAY use a common allocation between them.

A server encodes one of its assigned server IDs in any CID it generates using the relevant configuration.

4.3.1. Static Allocation

In the manual allocation method, the configuration agent assigns at least one server ID to each server.

When forwarding a packet with a long header and non-compliant DCID, load balancers MUST forward packets with long headers and non-compliant DCIDs using an arbitrary algorithm as specified in Section 4.2.

4.3.2. Dynamic Allocation

In the dynamic allocation method, the load balancer assigns server IDs dynamically so that configuration does not require bespoke server ID assignment. This also reduces linkability. However, it requires state at the load balancer that roughly scales with the number of connections, until the server ID codespace is exhausted.

4.3.2.1. Configuration Agent Actions

The configuration agent does not assign server IDs, but does configure a server ID length and an "LB timeout". The server ID MUST be at least one and no more than seven octets.

4.3.2.2. Load Balancer Actions

The load balancer maintains a table of all assigned server IDs and corresponding routing information, which is initialized empty. These tables are independent for each operating configuration.

The load balancer MUST keep track of the most recent observation of each server ID, in any sort of packet it forwards, in the table and delete the entries when the time since that observation exceeds the LB Timeout.

Note that when the load balancer’s table for a configuration is empty, all incoming DCIDs corresponding to that configuration are non-compliant by definition.

The handling of a non-compliant long-header packet depends on the reason for non-compliance. The load balancer MUST applyt this logic:

* If the config rotation bits do not match a known configuration, the load balancer routes the packet using an arbitrary algorithm (see Section 4.2).

* If there is a matching configuration, but the CID is not long enough to apply the algorithm, the load balancer skips the first octet of the CID and then reads a server ID from the following octets, up to the server ID length. If this server ID matches a known server ID for that configuration, it forwards the packet accordingly and takes no further action. If it does not match, it routes using an arbitrary algorithm and adds the new server ID to that server’s table entry.

* If the sole reason for non-compliance is that the server ID is not in the load balancer's table, the load balancer routes the packet with an arbitrary algorithm. It adds the decoded server ID to table entry for the server the algorithm chooses and forwards the packet accordingly.

4.3.2.3. Server actions

Each server maintains a list of server IDs assigned to it, initialized empty. For each SID, it records the last time it received any packet with an CID that encoded that SID.

Upon receipt of a packet with a client-generated DCID, the server MUST follow these steps in order:

* If the config rotation bits do not correspond to a known configuration, do not attempt to extract a server ID.

* If the DCID is not long enough to decode using the configured algorithm, extract a number of octets equal to the server ID length, beginning with the second octet. If the extracted value does not match a server ID in the server's list, add it to the list.

* If the DCID is long enough to decode but the server ID is not in the server's list, add it to the list.

After any possible SID is extracted, the server processes the packet normally.

When a server needs a new connection ID, it uses one of the server IDs in its list to populate the server ID field of that CID. It SHOULD vary this selection to reduce linkability within a connection.

After loading a new configuration or long periods of idleness, a server may not have any available SIDs. This is because an incoming packet may not the config rotation bits necessary to extract a server ID in accordance with the algorithm above. When required to generate a CID under these conditions, the server MUST generate CIDs using the 5-tuple routing codepoint (see Section 3.2. Note that these connections will not be robust to client address changes while they use this connection ID. For this reason, a server SHOULD retire these connection IDs and replace them with routable ones once it receives a client-generated CID that allows it to acquire a server ID. As, statistically, one in every four such CIDs can provide a server ID, this is typically a short interval.

If a server has not received a connection ID encoding a particular server ID within the LB timeout, it MUST retire any outstanding CIDs that use that server ID and cease generating any new ones.

A server SHOULD have a mechanism to stop using some server IDs if the list gets large relative to its share of the codepoint space, so that these allocations time out and are freed for reuse by servers that have recently joined the pool.

5. Routing Algorithms

Encryption in the algorithms below uses the AES-128-ECB cipher. Future standards could add new algorithms that use other ciphers to provide cryptographic agility in accordance with [RFC7696]. QUIC-LB implementations SHOULD be extensible to support new algorithms.

5.1. Plaintext CID Algorithm

The Plaintext CID Algorithm makes no attempt to obscure the mapping of connections to servers, significantly increasing linkability. The format is depicted in the figure below.

```
Plaintext CID {
  First Octet (8),
  Server ID (8..128),
  For Server Use (8..152-len(Server ID)),
}
```

Figure 3: Plaintext CID Format

5.1.1. Configuration Agent Actions

For static SID allocation, the server ID length is limited to 16 octets. There are no parameters specific to this algorithm.

5.1.2. Load Balancer Actions

On each incoming packet, the load balancer extracts consecutive octets, beginning with the second octet. These bytes represent the server ID.

5.1.3. Server Actions

The server chooses how many octets to reserve for its own use, which MUST be at least one octet.

When a server needs a new connection ID, it encodes one of its assigned server IDs in consecutive octets beginning with the second. All other bits in the connection ID, except for the first octet, MAY be set to any other value. These other bits SHOULD appear random to observers.

5.2. Stream Cipher CID Algorithm

The Stream Cipher CID algorithm provides cryptographic protection at the cost of additional per-packet processing at the load balancer to decrypt every incoming connection ID. The CID format is depicted below.

```
Stream Cipher CID {
  First Octet (8),
  Nonce (64..120),
  Encrypted Server ID (8..128-len(Nonce)),
  For Server Use (0..152-len(Nonce)-len(Encrypted Server ID)),
}
```

Figure 4: Stream Cipher CID Format

5.2.1. Configuration Agent Actions

The configuration agent assigns a server ID to every server in its pool, and determines a server ID length (in octets) sufficiently large to encode all server IDs, including potential future servers.

The configuration agent also selects a nonce length and an 16-octet AES-ECB key to use for connection ID decryption. The nonce length MUST be at least 8 octets and no more than 16 octets. The nonce length and server ID length MUST sum to 19 or fewer octets, but SHOULD sum to 15 or fewer to allow space for server use.

5.2.2. Load Balancer Actions

Upon receipt of a QUIC packet, the load balancer extracts as many of the earliest octets from the destination connection ID as necessary to match the nonce length. The server ID immediately follows.

The load balancer decrypts the nonce and the server ID using the following three pass algorithm:

* Pass 1: The load balancer decrypts the server ID using 128-bit AES Electronic Codebook (ECB) mode, much like QUIC header protection. The encrypted nonce octets are zero-padded to 16 octets. AES-ECB encrypts this encrypted nonce using its key to generate a mask which it applies to the encrypted server id. This provides an intermediate value of the server ID, referred to as server-id intermediate.

server_id_intermediate = encrypted_server_id ^ AES-ECB(key, padded-encrypted-nonce)

* Pass 2: The load balancer decrypts the nonce octets using 128-bit AES ECB mode, using the server-id intermediate as "nonce" for this pass. The server-id intermediate octets are zero-padded to 16 octets. AES-ECB encrypts this padded server-id intermediate using its key to generate a mask which it applies to the encrypted nonce. This provides the decrypted nonce value.

nonce = encrypted_nonce ^ AES-ECB(key, padded-server_id_intermediate)

* Pass 3: The load balancer decrypts the server ID using 128-bit AES ECB mode. The nonce octets are zero-padded to 16 octets. AES-ECB encrypts this nonce using its key to generate a mask which it applies to the intermediate server id. This provides the decrypted server ID.

server_id = server_id_intermediate ^ AES-ECB(key, padded-nonce)

For example, if the nonce length is 10 octets and the server ID length is 2 octets, the connection ID can be as small as 13 octets. The load balancer uses the the second through eleventh octets of the connection ID for the nonce, zero-pads it to 16 octets, uses xors the result with the twelfth and thirteenth octet. The result is padded with 14 octets of zeros and encrypted to obtain a mask that is xored with the nonce octets. Finally, the nonce octets are padded with six octets of zeros, encrypted, and the first two octets xored with the server ID octets to obtain the actual server ID.

This three-pass algorithm is a simplified version of the FFX algorithm, with the property that each encrypted nonce value depends on all server ID bits, and each encrypted server ID bit depends on all nonce bits and all server ID bits. This mitigates attacks against stream ciphers in which attackers simply flip encrypted server-ID bits.

The output of the decryption is the server ID that the load balancer uses for routing.

5.2.3. Server Actions

When generating a routable connection ID, the server writes arbitrary bits into its nonce octets, and its provided server ID into the server ID octets. Servers MAY opt to have a longer connection ID beyond the nonce and server ID. The additional bits MAY encode additional information, but SHOULD appear essentially random to observers.

If the decrypted nonce bits increase monotonically, that guarantees that nonces are not reused between connection IDs from the same server.

The server encrypts the server ID using exactly the algorithm as described in Section 5.2.2, performing the three passes in reverse order.

5.3. Block Cipher CID Algorithm

The Block Cipher CID Algorithm, by using a full 16 octets of plaintext and a 128-bit cipher, provides higher cryptographic protection and detection of non-compliant connection IDs. However, it also requires connection IDs of at least 17 octets, increasing overhead of client-to-server packets.

```
Block Cipher CID {
  First Octet (8),
  Encrypted Server ID (8..128),
  Encrypted Bits for Server Use (128-len(Encrypted Server ID)),
  Unencrypted Bits for Server Use (0..24),
}
```

Figure 5: Block Cipher CID Format

5.3.1. Configuration Agent Actions

If server IDs are statically allocated, the server ID length MUST be no more than 12 octets, to provide servers adequate entropy to generate unique CIDs.

The configuration agent also selects an 16-octet AES-ECB key to use for connection ID decryption.

5.3.2. Load Balancer Actions

Upon receipt of a QUIC packet, the load balancer reads the first octet to obtain the config rotation bits. It then decrypts the subsequent 16 octets using AES-ECB decryption and the chosen key.

The decrypted plaintext contains the server id and opaque server data in that order. The load balancer uses the server ID octets for routing.

5.3.3. Server Actions

When generating a routable connection ID, the server MUST choose a connection ID length between 17 and 20 octets. The server writes its server ID into the server ID octets and arbitrary bits into the remaining bits. These arbitrary bits MAY encode additional information, and MUST differ between connection IDs. Bits in the eighteenth, nineteenth, and twentieth octets SHOULD appear essentially random to observers. The first octet is reserved as described in Section 3.

The server then encrypts the second through seventeenth octets using the 128-bit AES-ECB cipher.

6. ICMP Processing

For protocols where 4-tuple load balancing is sufficient, it is straightforward to deliver ICMP packets from the network to the correct server, by reading the echoed IP and transport-layer headers to obtain the 4-tuple. When routing is based on connection ID, further measures are required, as most QUIC packets that trigger ICMP responses will only contain a client-generated connection ID that contains no routing information.

To solve this problem, load balancers MAY maintain a mapping of Client IP and port to server ID based on recently observed packets.

Alternatively, servers MAY implement the technique described in Section 14.4.1 of [QUIC-TRANSPORT] to increase the likelihood a Source Connection ID is included in ICMP responses to Path Maximum Transmission Unit (PMTU) probes. Load balancers MAY parse the echoed packet to extract the Source Connection ID, if it contains a QUIC long header, and extract the Server ID as if it were in a Destination CID.

7. Retry Service

When a server is under load, QUICv1 allows it to defer storage of connection state until the client proves it can receive packets at its advertised IP address. Through the use of a Retry packet, a token in subsequent client Initial packets, and transport parameters, servers verify address ownership and clients verify that there is no on-path attacker generating Retry packets.

A "Retry Service" detects potential Denial of Service attacks and handles sending of Retry packets on behalf of the server. As it is, by definition, literally an on-path entity, the service must communicate some of the original connection IDs back to the server so that it can pass client verification. It also must either verify the address itself (with the server trusting this verification) or make sure there is common context for the server to verify the address using a service-generated token.

There are two different mechanisms to allow offload of DoS mitigation to a trusted network service. One requires no shared state; the server need only be configured to trust a retry service, though this imposes other operational constraints. The other requires a shared key, but has no such constraints.

Retry services MUST forward all QUIC packets that are not of type Initial or 0-RTT. Other packet types might involve changed IP addresses or connection IDs, so it is not practical for Retry Services to identify such packets as valid or invalid.

7.1. Common Requirements

Regardless of mechanism, a retry service has an active mode, where it is generating Retry packets, and an inactive mode, where it is not, based on its assessment of server load and the likelihood an attack is underway. The choice of mode MAY be made on a per-packet or per-connection basis, through a stochastic process or based on client address.

A configuration agent MUST distribute a list of QUIC versions the Retry Service supports. It MAY also distribute either an "Allow-List" or a "Deny-List" of other QUIC versions. It MUST NOT distribute both an Allow-List and a Deny-List.

The Allow-List or Deny-List MUST NOT include any versions included for Retry Service Support.

The Configuration Agent MUST provide a means for the entity that controls the Retry Service to report its supported version(s) to the configuration Agent. If the entity has not reported this information, it MUST NOT activate the Retry Service and the configuration agent MUST NOT distribute configuration that activates it.

The configuration agent MAY delete versions from the final supported version list if policy does not require the Retry Service to operate on those versions.

The configuration Agent MUST provide a means for the entities that control servers behind the Retry Service to report either an Allow-List or a Deny-List.

If all entities supply Allow-Lists, the consolidated list MUST be the union of these sets. If all entities supply Deny-Lists, the consolidated list MUST be the intersection of these sets.

If entities provide a mixture of Allow-Lists and Deny-Lists, the consolidated list MUST be a Deny-List that is the intersection of all provided Deny-Lists and the inverses of all Allow-Lists.

If no entities that control servers have reported Allow-Lists or Deny-Lists, the default is a Deny-List with the null set (i.e., all unsupported versions will be admitted). This preserves the future extensibilty of QUIC.

A retry service MUST forward all packets for a QUIC version it does not support that are not on a Deny-List or absent from an Allow-List. Note that if servers support versions the retry service does not, this may increase load on the servers.

Note that future versions of QUIC might not have Retry packets, require different information in Retry, or use different packet type indicators.

7.2. No-Shared-State Retry Service

The no-shared-state retry service requires no coordination, except that the server must be configured to accept this service and know which QUIC versions the retry service supports. The scheme uses the first bit of the token to distinguish between tokens from Retry packets (codepoint '0') and tokens from NEW_TOKEN frames (codepoint '1').

7.2.1. Configuration Agent Actions

See Section 7.1.

7.2.2. Service Requirements

A no-shared-state retry service MUST be present on all paths from potential clients to the server. These paths MUST fail to pass QUIC traffic should the service fail for any reason. That is, if the service is not operational, the server MUST NOT be exposed to client traffic. Otherwise, servers that have already disabled their Retry capability would be vulnerable to attack.

The path between service and server MUST be free of any potential attackers. Note that this and other requirements above severely restrict the operational conditions in which a no-shared-state retry service can safely operate.

Retry tokens generated by the service MUST have the format below.

```
Non-Shared-State Retry Service Token {
  Token Type (1) = 0,
  ODCIL (7) = 8..20,
  RSCIL (8) = 0..20,
  Original Destination Connection ID (64..160),
  Retry Source Connection ID (0..160),
  Opaque Data (..),
}
```

Figure 6: Format of non-shared-state retry service tokens

The first bit of retry tokens generated by the service MUST be zero. The token has the following additional fields:

ODCIL: The length of the original destination connection ID from the triggering Initial packet. This is in cleartext to be readable for the server, but authenticated later in the token. The Retry Service SHOULD reject any token in which the value is less than 8.

RSCIL: The retry source connection ID length.

Original Destination Connection ID: This also in cleartext and authenticated later.

Retry Source Connection ID: This also in cleartext and authenticated later.

Opaque Data: This data MUST contain encrypted information that allows the retry service to validate the client's IP address, in accordance with the QUIC specification. It MUST also provide a cryptographically secure means to validate the integrity of the entire token.

Upon receipt of an Initial packet with a token that begins with '0', the retry service MUST validate the token in accordance with the QUIC specification.

In active mode, the service MUST issue Retry packets for all Client initial packets that contain no token, or a token that has the first bit set to '1'. It MUST NOT forward the packet to the server. The service MUST validate all tokens with the first bit set to '0'. If

successful, the service MUST forward the packet with the token intact. If unsuccessful, it MUST drop the packet. The Retry Service MAY send an Initial Packet containing a CONNECTION_CLOSE frame with the INVALID_TOKEN error code when dropping the packet.

Note that this scheme has a performance drawback. When the retry service is in active mode, clients with a token from a NEW_TOKEN frame will suffer a 1-RTT penalty even though its token provides proof of address.

In inactive mode, the service MUST forward all packets that have no token or a token with the first bit set to '1'. It MUST validate all tokens with the first bit set to '0'. If successful, the service MUST forward the packet with the token intact. If unsuccessful, it MUST either drop the packet or forward it with the token removed. The latter requires decryption and re-encryption of the entire Initial packet to avoid authentication failure. Forwarding the packet causes the server to respond without the original_destination_connection_id transport parameter, which preserves the normal QUIC signal to the client that there is an on-path attacker.

7.2.3. Server Requirements

A server behind a non-shared-state retry service MUST NOT send Retry packets for a QUIC version the retry service understands. It MAY send Retry for QUIC versions the Retry Service does not understand.

Tokens sent in NEW_TOKEN frames MUST have the first bit set to '1'.

If a server receives an Initial Packet with the first bit set to '1', it could be from a server-generated NEW_TOKEN frame and should be processed in accordance with the QUIC specification. If a server receives an Initial Packet with the first bit to '0', it is a Retry token and the server MUST NOT attempt to validate it. Instead, it MUST assume the address is validated and MUST extract the Original Destination Connection ID and Retry Source Connection ID, assuming the format described in Section 7.2.2.

7.3. Shared-State Retry Service

A shared-state retry service uses a shared key, so that the server can decode the service's retry tokens. It does not require that all traffic pass through the Retry service, so servers MAY send Retry packets in response to Initial packets that don't include a valid token.

Both server and service must have access to Universal time, though tight synchronization is unnecessary.

The tokens are protected using AES128-GCM AEAD, as explained in Section 7.3.1. All tokens, generated by either the server or retry service, MUST use the following format, which includes:

* A 96 bit unique token number transmitted in clear text, but protected as part of the AEAD associated data.

* An 8 bit token key identifier.

* A token body, encoding the Original Destination Connection ID, the Retry Source Connection ID, and the Timestamp, optionally followed by server specific Opaque Data.

The token protection uses an 128 bit representation of the source IP address from the triggering Initial packet. The client IP address is 16 octets. If an IPv4 address, the last 12 octets are zeroes.

If there is a Network Address Translator (NAT) in the server infrastructure that changes the client IP, the Retry Service MUST either be positioned behind the NAT, or the NAT must have the token key to rewrite the Retry token accordingly. Note also that a host that obtains a token through a NAT and then attempts to connect over a path that does not have an identically configured NAT will fail address validation.

The 96 bit unique token number is set to a random value using a cryptography- grade random number generator.

The token key identifier and the corresponding AEAD key and AEAD IV are provisioned by the configuration agent.

The token body is encoded as follows:

```
Shared-State Retry Service Token Body {
   ODCIL (8) = 0..20,
   RSCIL (8) = 0..20,
   [Port (16)],
   Original Destination Connection ID (0..160),
   Retry Source Connection ID (0..160),
   Timestamp (64),
   Opaque Data (..),
}
```

Figure 7: Body of shared-state retry service tokens

The token body has the following fields:

ODCIL: The original destination connection ID length. Tokens in NEW_TOKEN frames MUST set this field to zero.

RSCIL: The retry source connection ID length. Tokens in NEW_TOKEN frames MUST set this field to zero.

Port: The Source Port of the UDP datagram that triggered the Retry packet. This field MUST be present if and only if the ODCIL is greater than zero. This field is therefore always absent in tokens in NEW_TOKEN frames.

Original Destination Connection ID: The server or Retry Service copies this from the field in the client Initial packet.

Retry Source Connection ID: The server or Retry service copies this from the Source Connection ID of the Retry packet.

Timestamp: The Timestamp is a 64-bit integer, in network order, that expresses the expiration time of the token as a number of seconds in POSIX time (see Sec. 4.16 of [TIME_T]).

Opaque Data: The server may use this field to encode additional information, such as congestion window, RTT, or MTU. The Retry Service MUST have zero-length opaque data.

Some implementations of QUIC encode in the token the Initial Packet Number used by the client, in order to verify that the client sends the retried Initial with a PN larger that the triggering Initial. Such implementations will encode the Initial Packet Number as part of the opaque data. As tokens may be generated by the Service, servers MUST NOT reject tokens because they lack opaque data and therefore the packet number.

7.3.1. Token Protection with AEAD

On the wire, the token is presented as:

```
Shared-State Retry Service Token {
  Unique Token Number (96),
  Key Sequence (8),
  Encrypted Shared-State Retry Service Token Body (80..),
  AEAD Checksum (length depends on encryption algorithm),
}
```

Figure 8: Wire image of shared-state retry service tokens

The tokens are protected using AES128-GCM as follows:

* The token key and IV are retrieved using the Key Sequence.

* The nonce, N, is formed by combining the IV with the 96 bit unique token number. The 96 bits of the unique token number are left-padded with zeros to the size of the IV. The exclusive OR of the padded unique token number and the IV forms the AEAD nonce.

* The associated data is a formatted as a pseudo header by combining the cleartext part of the token with the IP address of the client.

```
Shared-State Retry Service Token Pseudoheader {
  IP Address (128),
  Unique Token Number (96),
  Key Sequence (8),
}
```

Figure 9: Psuedoheader for shared-state retry service tokens

* The input plaintext for the AEAD is the token body. The output ciphertext of the AEAD is transmitted in place of the token body.

* The AEAD Checksum is computed as part of the AEAD encryption process, and is verified during decryption.

7.3.2. Configuration Agent Actions

The configuration agent generates and distributes a "token key", a "token IV", a key sequence, and the information described in Section 7.1.

7.3.3. Service Requirements

In inactive mode, the Retry service forwards all packets without further inspection or processing.

Retry services MUST NOT issue Retry packets except where explicitly allowed below, to avoid sending a Retry packet in response to a Retry token.

When in active mode, the service MUST generate Retry tokens with the format described above when it receives a client Initial packet with no token.

The service SHOULD decrypt incoming tokens. The service SHOULD drop packets with unknown key sequence, or an AEAD checksum that does not match the expected value. (By construction, the AEAD checksum will only match if the client IP Address also matches.)

If the token checksum passes, and the ODCIL and RSCIL fields are both zero, then this is a NEW_TOKEN token generated by the server. Processing of NEW_TOKEN tokens is subtly different from Retry tokens, as described below.

The service SHOULD drop a packet containing a token where the ODCIL is greater than zero and less than the minimum number of octets for a client-generated CID (8 in QUIC version 1). The service also SHOULD drop a packet containing a token where the ODCIL is zero and RSCIL is nonzero.

If the Timestamp of a token points to time in the past, the token has expired; however, in order to allow for clock skew, it SHOULD NOT consider tokens to be expired if the Timestamp encodes a few seconds in the past. An active Retry service SHOULD drop packets with expired tokens. If a NEW_TOKEN token, the service MUST generate a Retry packet in response. It MUST NOT generate a Retry packet in response to an expired Retry token.

If a Retry token, the service SHOULD drop packets where the port number encoded in the token does not match the source port in the encapsulating UDP header.

All other packets SHOULD be forwarded to the server.

7.3.4. Server Requirements

When issuing Retry or NEW_TOKEN tokens, the server MUST include the client IP address in the authenticated data as specified in Section 7.3.1. The ODCIL and RSCIL fields are zero for NEW_TOKEN tokens, making them easily distinguishable from Retry tokens.

The server MUST validate all tokens that arrive in Initial packets, as they may have bypassed the Retry service.

For Retry tokens that follow the format above, servers SHOULD use the timestamp field to apply its expiration limits for tokens. This need not be precisely synchronized with the retry service. However, servers MAY allow retry tokens marked as being a few seconds in the past, due to possible clock synchronization issues.

After decrypting the token, the server uses the corresponding fields to populate the original_destination_connection_id transport parameter, with a length equal to ODCIL, and the retry_source_connection_id transport parameter, with length equal to RSCIL.

For QUIC versions the service does not support, the server MAY use any token format.

As discussed in [QUIC-TRANSPORT], a server MUST NOT send a Retry packet in response to an Initial packet that contains a retry token.

8. Configuration Requirements

QUIC-LB requires common configuration to synchronize understanding of encodings and guarantee explicit consent of the server.

The load balancer and server MUST agree on a routing algorithm, server ID allocation method, and the relevant parameters for that algorithm.

All algorithms require a server ID length. If server IDs are statically allocated, the load balancer MUST receive the full table of mappings, and each server must receive its assigned SID(s), from the configuration agent.

For Stream Cipher CID Routing, the servers and load balancer also MUST have a common understanding of the key and nonce length.

For Block Cipher CID Routing, the servers and load balancer also MUST have a common understanding of the key.

Note that server IDs are opaque bytes, not integers, so there is no notion of network order or host order.

A server configuration MUST specify if the first octet encodes the CID length. Note that a load balancer does not need the CID length, as the required bytes are present in the QUIC packet.

A full QUIC-LB server configuration MUST also specify the supported QUIC versions of any Retry Service. If a shared-state service, the server also must have the token key.

A non-shared-state Retry Service need only be configured with the QUIC versions it supports, and an Allow- or Deny-List. A shared-state Retry Service also needs the token key, and to be aware if a NAT sits between it and the servers.

Appendix A provides a YANG Model of the a full QUIC-LB configuration.

9. Additional Use Cases

This section discusses considerations for some deployment scenarios not implied by the specification above.

9.1. Load balancer chains

Some network architectures may have multiple tiers of low-state load balancers, where a first tier of devices makes a routing decision to the next tier, and so on, until packets reach the server. Although QUIC-LB is not explicitly designed for this use case, it is possible to support it.

If each load balancer is assigned a range of server IDs that is a subset of the range of IDs assigned to devices that are closer to the client, then the first devices to process an incoming packet can extract the server ID and then map it to the correct forwarding address. Note that this solution is extensible to arbitrarily large numbers of load-balancing tiers, as the maximum server ID space is quite large.

9.2. Moving connections between servers

Some deployments may transparently move a connection from one server to another. The means of transferring connection state between servers is out of scope of this document.

To support a handover, a server involved in the transition could issue CIDs that map to the new server via a NEW_CONNECTION_ID frame, and retire CIDs associated with the new server using the "Retire Prior To" field in that frame.

Alternately, if the old server is going offline, the load balancer could simply map its server ID to the new server’s address.

10. Version Invariance of QUIC-LB

Non-shared-state Retry Services are inherently dependent on the format (and existence) of Retry Packets in each version of QUIC, and so Retry Service configuration explicitly includes the supported QUIC versions.

The server ID encodings, and requirements for their handling, are designed to be QUIC version independent (see [QUIC-INVARIANTS]). A QUIC-LB load balancer will generally not require changes as servers deploy new versions of QUIC. However, there are several unlikely future design decisions that could impact the operation of QUIC-LB.

The maximum Connection ID length could be below the minimum necessary for one or more encoding algorithms.

Section 4.1 provides guidance about how load balancers should handle non-compliant DCIDs. This guidance, and the implementation of an algorithm to handle these DCIDs, rests on some assumptions:

* Incoming short headers do not contain DCIDs that are client-generated.

* The use of client-generated incoming DCIDs does not persist beyond a few round trips in the connection.

* While the client is using DCIDs it generated, some exposed fields (IP address, UDP port, client-generated destination Connection ID) remain constant for all packets sent on the same connection.

* Dynamic server ID allocation is dependent on client-generated Destination CIDs in Initial Packets being at least 8 octets in length. If they are not, the load balancer may not be able to extract a valid server ID to add to its table. Configuring a shorter server ID length can increase robustness to a change.

While this document does not update the commitments in [QUIC-INVARIANTS], the additional assumptions are minimal and narrowly scoped, and provide a likely set of constants that load balancers can use with minimal risk of version- dependence.

If these assumptions are invalid, this specification is likely to lead to loss of packets that contain non-compliant DCIDs, and in extreme cases connection failure.

Some load balancers might inspect elements of the Server Name Indication (SNI) extension in the TLS Client Hello to make a routing decision. Note that the format and cryptographic protection of this information may change in future versions or extensions of TLS or QUIC, and therefore this functionality is inherently not version-invariant.

11. Security Considerations

QUIC-LB is intended to prevent linkability. Attacks would therefore attempt to subvert this purpose.

Note that the Plaintext CID algorithm makes no attempt to obscure the server mapping, and therefore does not address these concerns. It exists to allow consistent CID encoding for compatibility across a network infrastructure, which makes QUIC robust to NAT rebinding. Servers that are running the Plaintext CID algorithm SHOULD only use it to generate new CIDs for the Server Initial Packet and SHOULD NOT send CIDs in QUIC NEW_CONNECTION_ID frames, except that it sends one new Connection ID in the event of config rotation Section 3.1. Doing so might falsely suggest to the client that said CIDs were generated in a secure fashion.

A linkability attack would find some means of determining that two connection IDs route to the same server. As described above, there is no scheme that strictly prevents linkability for all traffic patterns, and therefore efforts to frustrate any analysis of server ID encoding have diminishing returns.

11.1. Attackers not between the load balancer and server

Any attacker might open a connection to the server infrastructure and aggressively simulate migration to obtain a large sample of IDs that map to the same server. It could then apply analytical techniques to try to obtain the server encoding.

The Stream and Block Cipher CID algorithms provide robust protection against any sort of linkage. The Plaintext CID algorithm makes no attempt to protect this encoding.

Were this analysis to obtain the server encoding, then on-path observers might apply this analysis to correlating different client IP addresses.

11.2. Attackers between the load balancer and server

Attackers in this privileged position are intrinsically able to map two connection IDs to the same server. The QUIC-LB algorithms do prevent the linkage of two connection IDs to the same individual connection if servers make reasonable selections when generating new IDs for that connection.

11.3. Multiple Configuration IDs

During the period in which there are multiple deployed configuration IDs (see Section 3.1), there is a slight increase in linkability. The server space is effectively divided into segments with CIDs that have different config rotation bits. Entities that manage servers SHOULD strive to minimize these periods by quickly deploying new configurations across the server pool.

11.4. Limited configuration scope

A simple deployment of QUIC-LB in a cloud provider might use the same global QUIC-LB configuration across all its load balancers that route to customer servers. An attacker could then simply become a customer, obtain the configuration, and then extract server IDs of other customers' connections at will.

To avoid this, the configuration agent SHOULD issue QUIC-LB configurations to mutually distrustful servers that have different keys for encryption algorithms. The load balancers can distinguish these configurations by external IP address, or by assigning different values to the config rotation bits (Section 3.1). Note that either solution has a privacy impact; see Section 11.3.

These techniques are not necessary for the plaintext algorithm, as it does not attempt to conceal the server ID.

11.5. Stateless Reset Oracle

Section 21.9 of [QUIC-TRANSPORT] discusses the Stateless Reset Oracle attack. For a server deployment to be vulnerable, an attacking client must be able to cause two packets with the same Destination CID to arrive at two different servers that share the same cryptographic context for Stateless Reset tokens. As QUIC-LB requires deterministic routing of DCIDs over the life of a connection, it is a sufficient means of avoiding an Oracle without additional measures.

11.6. Connection ID Entropy

The Stream Cipher and Block Cipher algorithms need to generate different cipher text for each generated Connection ID instance to protect the Server ID. To do so, at least four octets of the Block Cipher CID and at least eight octets of the Stream Cipher CID are reserved for a nonce that, if used only once, will result in unique cipher text for each Connection ID.

If servers simply increment the nonce by one with each generated connection ID, then it is safe to use the existing keys until any server's nonce counter exhausts the allocated space and rolls over to zero. Whether or not it implements this method, the server MUST NOT reuse a nonce until it switches to a configuration with new keys.

Configuration agents SHOULD implement an out-of-band method to discover when servers are in danger of exhausting their nonce space, and SHOULD respond by issuing a new configuration. A server that has exhausted its nonces MUST either switch to a different configuration, or if none exists, use the 4-tuple routing config rotation codepoint.

11.7. Shared-State Retry Keys

The Shared-State Retry Service defined in Section 7.3 describes the format of retry tokens or new tokens protected and encrypted using AES128-GCM. Each token includes a 96 bit randomly generated unique token number, and an 8 bit identifier of the AES-GCM encryption key. There are three important security considerations for these tokens:

* An attacker that obtains a copy of the encryption key will be able to decrypt and forge tokens.

* Attackers may be able to retrieve the key if they capture a sufficently large number of retry tokens encrypted with a given key.

* Confidentiality of the token data will fail if separate tokens reuse the same 96 bit unique token number and the same key.

To protect against disclosure of keys to attackers, service and servers MUST ensure that the keys are stored securely. To limit the consequences of potential exposures, the time to live of any given key should be limited.

Section 6.6 of [QUIC-TLS] states that "Endpoints MUST count the number of encrypted packets for each set of keys. If the total number of encrypted packets with the same key exceeds the confidentiality limit for the selected AEAD, the endpoint MUST stop using those keys." It goes on with the specific limit: "For AEAD_AES_128_GCM and AEAD_AES_256_GCM, the confidentiality limit is 2^23 encrypted packets; see Appendix B.1." It is prudent to adopt the same limit here, and configure the service in such a way that no more than 2^23 tokens are generated with the same key.

In order to protect against collisions, the 96 bit unique token numbers should be generated using a cryptographically secure pseudorandom number generator (CSPRNG), as specified in Appendix C.1

of the TLS 1.3 specification [RFC8446]. With proper random numbers, if fewer than 2^40 tokens are generated with a single key, the risk of collisions is lower than 0.001%.

12. IANA Considerations

There are no IANA requirements.

13. References

13.1. Normative References

[QUIC-INVARIANTS]
Thomson, M., "Version-Independent Properties of QUIC", Work in Progress, Internet-Draft, draft-ietf-quic-invariants-13, 14 January 2021, <http://www.ietf.org/internet-drafts/draft-ietf-quic-invariants-13.txt>.

[QUIC-TRANSPORT]
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", Work in Progress, Internet-Draft, draft-ietf-quic-transport-34, 14 January 2021, <http://www.ietf.org/internet-drafts/draft-ietf-quic-transport-34.txt>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

[TIME_T] "Open Group Standard: Vol. 1: Base Definitions, Issue 7", IEEE Std 1003.1 , 2018, <http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap04.html#tag_04_16>.

13.2. Informative References

[QUIC-TLS] Thomson, M. and S. Turner, "Using TLS to Secure QUIC", Work in Progress, Internet-Draft, draft-ietf-quic-tls-34, 14 January 2021, <http://www.ietf.org/internet-drafts/draft-ietf-quic-tls-34.txt>.

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

[RFC6020] Bjorklund, M., Ed., "YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)", RFC 6020, DOI 10.17487/RFC6020, October 2010, <https://www.rfc-editor.org/info/rfc6020>.

[RFC7696] Housley, R., "Guidelines for Cryptographic Algorithm Agility and Selecting Mandatory-to-Implement Algorithms", BCP 201, RFC 7696, DOI 10.17487/RFC7696, November 2015, <https://www.rfc-editor.org/info/rfc7696>.

[RFC8340] Bjorklund, M. and L. Berger, Ed., "YANG Tree Diagrams", BCP 215, RFC 8340, DOI 10.17487/RFC8340, March 2018, <https://www.rfc-editor.org/info/rfc8340>.

Appendix A. QUIC-LB YANG Model

This YANG model conforms to [RFC6020] and expresses a complete QUIC-LB configuration.

```
module ietf-quic-lb {
  yang-version "1.1";
  namespace "urn:ietf:params:xml:ns:yang:ietf-quic-lb";
  prefix "quic-lb";

  import ietf-yang-types {
    prefix yang;
    reference
      "RFC 6991: Common YANG Data Types.";
  }

  import ietf-inet-types {
    prefix inet;
    reference
      "RFC 6991: Common YANG Data Types.";
  }

  organization
    "IETF QUIC Working Group";

  contact
    "WG Web:   <http://datatracker.ietf.org/wg/quic>
     WG List:  <quic@ietf.org>

     Authors: Martin Duke (martin.h.duke at gmail dot com)
              Nick Banks (nibanks at microsoft dot com)";

  description
    "This module enables the explicit cooperation of QUIC servers with
```

```
    trusted intermediaries without breaking important protocol features.

    Copyright (c) 2021 IETF Trust and the persons identified as
    authors of the code.  All rights reserved.

    Redistribution and use in source and binary forms, with or
    without modification, is permitted pursuant to, and subject to
    the license terms contained in, the Simplified BSD License set
    forth in Section 4.c of the IETF Trust's Legal Provisions
    Relating to IETF Documents
    (https://trustee.ietf.org/license-info).

    This version of this YANG module is part of RFC XXXX
    (https://www.rfc-editor.org/info/rfcXXXX); see the RFC itself
    for full legal notices.

    The key words 'MUST', 'MUST NOT', 'REQUIRED', 'SHALL', 'SHALL
    NOT', 'SHOULD', 'SHOULD NOT', 'RECOMMENDED', 'NOT RECOMMENDED',
    'MAY', and 'OPTIONAL' in this document are to be interpreted as
    described in BCP 14 (RFC 2119) (RFC 8174) when, and only when,
    they appear in all capitals, as shown here.";

  revision "2021-01-29" {
    description
      "Initial Version";
    reference
      "RFC XXXX, QUIC-LB: Generating Routable QUIC Connection IDs";
  }

  container quic-lb {
    presence "The container for QUIC-LB configuration.";

    description
      "QUIC-LB container.";

    typedef quic-lb-key {
      type yang:hex-string {
        length 47;
      }
      description
        "This is a 16-byte key, represented with 47 bytes";
    }

    list cid-configs {
      key "config-rotation-bits";
      description
        "List up to three load balancer configurations";
```

```
      leaf config-rotation-bits {
        type uint8 {
          range "0..2";
        }
        mandatory true;
        description
          "Identifier for this CID configuration.";
      }

      leaf first-octet-encodes-cid-length {
        type boolean;
        default false;
        description
          "If true, the six least significant bits of the first CID
           octet encode the CID length minus one.";
      }

      leaf cid-key {
        type quic-lb-key;
        description
          "Key for encrypting the connection ID. If absent, the
           configuration uses the Plaintext algorithm.";
      }

      leaf nonce-length {
        type uint8 {
          range "8..16";
        }
        must '(../cid-key)' {
          error-message "nonce-length only valid if cid-key is set";
        }
        description
          "Length, in octets, of the nonce. If absent when cid-key is
           present, the configuration uses the Block Cipher Algorithm.
           If present along with cid-key, the configurationuses the
           Stream Cipher Algorithm.";
      }

      leaf lb-timeout {
        type uint32;
        description
          "Existence means the configuration uses dynamic Server ID allocation.
           Time (in seconds) to keep a server ID allocation if no packets with
           that server ID arrive.";
      }

      leaf server-id-length {
        type uint8 {
```

```
          range "1..18";
        }
        must '(../lb-timeout and . <= 7) or
               (not(../lb-timeout) and
                (not(../cid-key) and . <= 16) or
                ((../nonce-length) and . <= (19 - ../nonce-length)) or
                ((../cid-key) and not(../nonce-length) and . <= 12))' {
          error-message
            "Server ID length too long for routing algorithm and server ID
             allocation method";
        }
        mandatory true;
        description
          "Length (in octets) of a server ID. Further range-limited
           by sid-allocation, cid-key, and nonce-length.";
      }

      list server-id-mappings {
        when "not(../lb-timeout)";
        key "server-id";
        description "Statically allocated Server IDs";

        leaf server-id {
          type yang:hex-string;
          must "string-length(.) = 3 * ../../server-id-length - 1";
          mandatory true;
          description
            "An allocated server ID";
        }

        leaf server-address {
          type inet:ip-address;
          mandatory true;
          description
            "Destination address corresponding to the server ID";
        }
      }
    }

    container retry-service-config {
      description
        "Configuration of Retry Service. If supported-versions is empty, there
         is no retry service. If token-keys is empty, it uses the non-shared-
         state service. If present, it uses shared-state tokens.";

      leaf-list supported-versions {
        type uint32;
        description
```

```
          "QUIC versions that the retry service supports. If empty, there
           is no retry service.";
      }

      leaf unsupported-version-default {
        type enumeration {
          enum allow {
            description "Unsupported versions admitted by default";
          }
          enum deny {
            description "Unsupported versions denied by default";
          }
        }
        default allow;
        description
          "Are unsupported versions not in version-exceptions allowed
           or denied?";
      }

      leaf-list version-exceptions {
        type uint32;
        description
          "Exceptions to the default-deny or default-allow rule.";
      }

      list token-keys {
        key "key-sequence-number";
        description
          "list of active keys, for key rotation purposes. Existence implies
           shared-state format";

        leaf key-sequence-number {
          type uint8;
          mandatory true;
          description
            "Identifies the key used to encrypt the token";
        }

        leaf token-key {
          type quic-lb-key;
          mandatory true;
          description
            "16-byte key to encrypt the token";
        }

        leaf token-iv {
          type yang:hex-string {
            length 23;
```

```
          }
          mandatory true;
          description
            "8-byte IV to encrypt the token, encoded in 23 bytes";
        }
      }
    }
  }
}
```

A.1. Tree Diagram

This summary of the YANG model uses the notation in [RFC8340].

```
 module: ietf-quic-lb
   +--rw quic-lb
      +--rw cid-configs*
      |       [config-rotation-bits]
      |  +--rw config-rotation-bits             uint8
      |  +--rw first-octet-encodes-cid-length?  boolean
      |  +--rw cid-key?                         yang:hex-string
      |  +--rw nonce-length?                    uint8
      |  +--rw lb-timeout?                      uint32
      |  +--rw server-id-length                 uint8
      |  +--rw server-id-mappings*?
      |  |       [server-id]
      |  |  +--rw server-id                     yang:hex-string
      |  |  +--rw server-address                inet:ip-address
      +--ro retry-service-config
      |  +--rw supported-versions*
      |  |  +--rw version                       uint32
      |  +--rw unsupported-version-default      enumeration {allow deny}
      |  +--rw version-exceptions*
      |  |  +--rw version                       uint32
      |  +--rw token-keys*?
      |  |       [key-sequence-number]
      |  |  +--rw key-sequence-number           uint8
      |  |  +--rw token-key                     yang:hex-string
      |  |  +--rw token-iv                      yang:hex-string
```

Appendix B. Load Balancer Test Vectors

Each section of this draft includes multiple sets of load balancer configuration, each of which has five examples of server ID and server use bytes and how they are encoded in a CID.

In some cases, there are no server use bytes. Note that, for simplicity, the first octet bits used for neither config rotation nor length self-encoding are random, rather than listed in the server use field. Therefore, a server implementation using these parameters may generate CIDs with a slightly different first octet.

This section uses the following abbreviations:

cid	Connection ID
cr_bits	Config Rotation Bits
LB	Load Balancer
sid	Server ID
sid_len	Server ID length
su	Server Use Bytes

All values except length_self_encoding and sid_len are expressed in hexidecimal format.

B.1. Plaintext Connection ID Algorithm

LB configuration: cr_bits 0x0 length_self_encoding: y sid_len 1

cid 01be sid be su
cid 0221b7 sid 21 su b7
cid 03cadfd8 sid ca su dfd8
cid 041e0c9328 sid 1e su 0c9328
cid 050c8f6d9129 sid 0c su 8f6d9129

LB configuration: cr_bits 0x0 length_self_encoding: n sid_len 2

cid 02aab0 sid aab0 su
cid 3ac4b106 sid c4b1 su 06
cid 08bd3cf4a0 sid bd3c su f4a0
cid 3771d59502d6 sid 71d5 su 9502d6
cid 1d57dee8b888f3 sid 57de su e8b888f3

LB configuration: cr_bits 0x0 length_self_encoding: y sid_len 3

cid 0336c976 sid 36c976 su
cid 04aa291806 sid aa2918 su 06
cid 0586897bd8b6 sid 86897b su d8b6
cid 063625bcae4de0 sid 3625bc su ae4de0
cid 07966fb1f3cb535f sid 966fb1 su f3cb535f

LB configuration: cr_bits 0x0 length_self_encoding: n sid_len 4

cid 185172fab8 sid 5172fab8 su
cid 2eb7ff2c9297 sid b7ff2c92 su 97
cid 14f3eb3dd3edbe sid f3eb3dd3 su edbe
cid 3feb31cece744b74 sid eb31cece su 744b74
cid 06b9f34c353ce23bb5 sid b9f34c35 su 3ce23bb5

LB configuration: cr_bits 0x0 length_self_encoding: y sid_len 5

cid 05bdcd8d0b1d sid bdcd8d0b1d su
cid 06aee673725a63 sid aee673725a su 63
cid 07bbf338ddbf37f4 sid bbf338ddbf su 37f4
cid 08fbbca64c26756840 sid fbbca64c26 su 756840
cid 09e7737c495b93894e34 sid e7737c495b su 93894e34

B.2. Stream Cipher Connection ID Algorithm

In each case below, the server is using a plain text nonce value of zero.

```
LB configuration: cr_bits 0x0 length_self_encoding: y nonce_len 12 sid_len 1
    key 4d9d0fd25a25e7f321ef464e13f9fa3d

cid 0d69fe8ab8293680395ae256e89c sid c5 su
cid 0e420d74ed99b985e10f5073f43027 sid d5 su 27
cid 0f380f440c6eefd3142ee776f6c16027 sid 10 su 6027
cid 1020607efbe82049ddbf3a7c3d9d32604d sid 3c su 32604d
cid 11e132d12606a1bb0fa17e1caef00ec54c10 sid e3 su 0ec54c10

LB configuration: cr_bits 0x0 length_self_encoding: n nonce_len 12 sid_len 2
    key 49e1cec7fd264b1f4af37413baf8ada9

cid 3d3a5e1126414271cc8dc2ec7c8c15 sid f7fe su
cid 007042539e7c5f139ac2adfbf54ba748 sid eaf4 su 48
cid 2bc125dd2aed2aafacf59855d99e029217 sid e880 su 9217
cid 3be6728dc082802d9862c6c8e4dda3d984d8 sid 62c6 su d984d8
cid 1afe9c6259ad350fc7bad28e0aeb2e8d4d4742 sid 8502 su 8d4d4742

LB configuration: cr_bits 0x0 length_self_encoding: y nonce_len 14 sid_len 3
    key 2c70df0b399bd33a7335523dcdb884ad

cid 11d62e8670565cd30b552edff6782ff5a740 sid d794bb su
cid 12c70e481f49363cabd9370d1fd5012c12bca5 sid 2cbd5d su a5
cid 133b95dfd8ad93566782f8424df82458069fc9e9 sid d126cd su c9e9
cid 13ac6ffcd635532ab60370306c7ee572d6b6e795 sid 539e42 su e795
cid 1383ed07a9700777ff450bb39bb9c1981266805c sid 9094dd su 805c

LB configuration: cr_bits 0x0 length_self_encoding: n nonce_len 12 sid_len 4
    key 2297b8a95c776cf9c048b76d9dc27019

cid 32873890c3059ca62628089439c44c1f84 sid 7398d8ca su
cid 1ff7c7d7b9823954b178636c99a7dc93ac83 sid 9655f091 su 83
cid 31044000a5ebb3bf2fa7629a17f2c78b077c17 sid 8b035fc6 su 7c17
cid 1791bd28c66721e8fea0c6f34fd2d8e663a6ef70 sid 6672e0e2 su a6ef70
cid 3df1d90ad5ccd5f8f475f040e90aeca09ec9839d sid b98b1fff su c9839d

LB configuration: cr_bits 0x0 length_self_encoding: y nonce_len 8 sid_len 5
    key 484b2ed942d9f4765e45035da3340423

cid 0da995b7537db605bfd3a38881ae sid 391a7840dc su
cid 0ed8d02d55b91d06443540d1bf6e98 sid 10f7f7b284 su 98
cid 0f3f74be6d46a84ccb1fd1ee92cdeaf2 sid 0606918fc0 su eaf2
cid 1045626dbf20e03050837633cc5650f97c sid e505eea637 su 50f97c
cid 11bb9a17f691ab446a938427febbeb593eaa sid 99343a2a96 su eb593eaa
```

B.3. Block Cipher Connection ID Algorithm

```
LB configuration: cr_bits 0x0 length_self_encoding: y sid_len 1
    key 411592e4160268398386af84ea7505d4

cid 10564f7c0df399f6d93bdddb1a03886f25 sid 23 su 05231748a80884ed58007847eb9fd0
cid 10d5c03f9dd765d73b3d8610b244f74d02 sid 15 su 76cd6b6f0d3f0b20fc8e633e3a05f3
cid 108ca55228ab23b92845341344a2f956f2 sid 64 su 65c0ce170a9548717498b537cb8790
cid 10e73f3d034aef2f6f501e3a7693d6270a sid 07 su f9ad10c84cc1e89a2492221d74e707
cid 101a6ce13d48b14a77ecfd365595ad2582 sid 6c su 76ce4689b0745b956ef71c2608045d

LB configuration: cr_bits 0x0 length_self_encoding: n sid_len 2
    key 92ce44aecd636aeeff78da691ef48f77

cid 20aa09bc65ed52b1ccd29feb7ef995d318 sid a52f su 99278b92a86694ff0ecd64bc2f73
cid 30b8dbef657bd78a2f870e93f9485d5211 sid 6c49 su 7381c8657a388b4e9594297afe96
cid 043a8137331eacd2e78383279b202b9a6d sid 4188 su 5ac4b0e0b95f4e7473b49ee2d0dd
cid 3ba71ea2bcf0ab95719ab59d3d7fde770d sid 8ccc su 08728807605db25f2ca88be08e0f
cid 37ef1956b4ec354f40dc68336a23d42b31 sid c89d su 5a3ccd1471caa0de221ad6c185c0

LB configuration: cr_bits 0x0 length_self_encoding: y sid_len 3
    key 5c49cb9265efe8ae7b1d3886948b0a34

cid 10efcffc161d232d113998a49b1dbc4aa0 sid 0690b3 su 958fc9f38fe61b83881b2c5780
cid 10fc13bdbcb414ba90e391833400c19505 sid 031ac3 su 9a55e1e1904e780346fcc32c3c
cid 10d3cc1efaf5dc52c7a0f6da2746a8c714 sid 572d3a su ff2ec9712664e7174dc03ca3f8
cid 107edf37f6788e33c0ec7758a485215f2b sid 562c25 su 02c5a5dcbea629c3840da5f567
cid 10bc28da122582b7312e65aa096e9724fc sid 2fa4f0 su 8ae8c666bfc0fc364ebfd06b9a

LB configuration: cr_bits 0x0 length_self_encoding: n sid_len 4
    key e787a3a491551fb2b4901a3fa15974f3

cid 26125351da12435615e3be6b16fad35560 sid 0cb227d3 su 65b40b1ab54e05bff55db046
cid 14de05fc84e41b611dfbe99ed5b1c9d563 sid 6a0f23ad su d73bee2f3a7e72b3ffea52d9
cid 1306052c3f973db87de6d7904914840ff1 sid ca21402d su 5829465f7418b56ee6ada431
cid 1d202b5811af3e1dba9ea2950d27879a92 sid b14e1307 su 4902aba8b23a5f24616df3cf
cid 26538b78efc2d418539ad1de13ab73e477 sid a75e0148 su 0040323f1854e75aeb449b9f

LB configuration: cr_bits 0x0 length_self_encoding: y sid_len 5
    key d5a6d7824336fbe0f25d28487cdda57c

cid 10a2794871aadb20ddf274a95249e57fde sid 82d3b0b1a1 su 0935471478c2edb8120e60
cid 108122fe80a6e546a285c475a3b8613ec9 sid fbcc902c9d su 59c47946882a9a93981c15
cid 104d227ad9dd0fef4c8cb6eb75887b6ccc sid 2808e22642 su 2a7ef40e2c7e17ae40b3fb
cid 10b3f367d8627b36990a28d67f50b97846 sid 5e018f0197 su 2289cae06a566e5cb6cfa4
cid 1024412bfe25f4547510204bdda6143814 sid 8a8dd3d036 su 4b12933a135e5eaaebc6fd
```

Appendix C. Acknowledgments

The authors would like to thank Christian Huitema and Ian Swett for their major design contributions.

Manasi Deval, Erik Fuller, Toma Gavrichenkov, Jana Iyengar, Subodh Iyengar, Ladislav Lhotka, Jan Lindblad, Ling Tao Nju, Kazuho Oku, Udip Pant, Martin Thomson, Dmitri Tikhonov, Victor Vasiliev, and William Zeng Ke all provided useful input to this document.

Appendix D. Change Log

RFC Editor's Note: Please remove this section prior to publication of a final version of this document.

D.1. since draft-ietf-quic-load-balancers-05

* Added low-config CID for further discussion
* Complete revision of shared-state Retry Token
* Added YANG model
* Updated configuration limits to ensure CID entropy
* Switched to notation from quic-transport

D.2. since draft-ietf-quic-load-balancers-04

* Rearranged the shared-state retry token to simplify token processing
* More compact timestamp in shared-state retry token
* Revised server requirements for shared-state retries
* Eliminated zero padding from the test vectors
* Added server use bytes to the test vectors
* Additional compliant DCID criteria

D.3. since-draft-ietf-quic-load-balancers-03

* Improved Config Rotation text
* Added stream cipher test vectors

* Deleted the Obfuscated CID algorithm

D.4. since-draft-ietf-quic-load-balancers-02

* Replaced stream cipher algorithm with three-pass version
* Updated Retry format to encode info for required TPs
* Added discussion of version invariance
* Cleaned up text about config rotation
* Added Reset Oracle and limited configuration considerations
* Allow dropped long-header packets for known QUIC versions

D.5. since-draft-ietf-quic-load-balancers-01

* Test vectors for load balancer decoding
* Deleted remnants of in-band protocol
* Light edit of Retry Services section
* Discussed load balancer chains

D.6. since-draft-ietf-quic-load-balancers-00

* Removed in-band protocol from the document

D.7. Since draft-duke-quic-load-balancers-06

* Switch to IETF WG draft.

D.8. Since draft-duke-quic-load-balancers-05

* Editorial changes
* Made load balancer behavior independent of QUIC version
* Got rid of token in stream cipher encoding, because server might not have it
* Defined "non-compliant DCID" and specified rules for handling them.
* Added psuedocode for config schema

D.9. Since draft-duke-quic-load-balancers-04

* Added standard for retry services

D.10. Since draft-duke-quic-load-balancers-03

* Renamed Plaintext CID algorithm as Obfuscated CID
* Added new Plaintext CID algorithm
* Updated to allow 20B CIDs
* Added self-encoding of CID length

D.11. Since draft-duke-quic-load-balancers-02

* Added Config Rotation
* Added failover mode
* Tweaks to existing CID algorithms
* Added Block Cipher CID algorithm
* Reformatted QUIC-LB packets

D.12. Since draft-duke-quic-load-balancers-01

* Complete rewrite
* Supports multiple security levels
* Lightweight messages

D.13. Since draft-duke-quic-load-balancers-00

* Converted to markdown
* Added variable length connection IDs

Authors' Addresses

Martin Duke
F5 Networks, Inc.

Email: martin.h.duke@gmail.com

Nick Banks
Microsoft

Email: nibanks@microsoft.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: 23 October 2021

M. Kuehlewind
Ericsson
B. Trammell
Google Switzerland GmbH

21 April 2021

Manageability of the QUIC Transport Protocol
draft-ietf-quic-manageability-11

Abstract

This document discusses manageability of the QUIC transport protocol, focusing on the implications of QUIC's design and wire image on network operations involving QUIC traffic. Its intended audience is network operators and equipment vendors who rely on the use of transport-aware network functions.

Status of This Memo

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

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

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

This Internet-Draft will expire on 23 October 2021.

Table of Contents

1. Introduction

QUIC [QUIC-TRANSPORT] is a new transport protocol that is encapsulated in UDP. QUIC integrates TLS [QUIC-TLS] to encrypt all payload data and most control information. QUIC version 1 was designed primarily as a transport for HTTP, with the resulting protocol being known as HTTP/3 [QUIC-HTTP].

This document provides guidance for network operations that manage QUIC traffic. This includes guidance on how to interpret and utilize information that is exposed by QUIC to the network, requirements and assumptions of the QUIC design with respect to network treatment, and a description of how common network management practices will be impacted by QUIC.

QUIC is an end-to-end transport protocol. No information in the protocol header, even that which can be inspected, is meant to be mutable by the network. This is achieved through integrity protection of the wire image [WIRE-IMAGE]. Encryption of most control signaling means that less information is visible to the network than is the case with TCP.

Integrity protection can also simplify troubleshooting, because none of the nodes on the network path can modify transport layer information. However, it does imply that in-network operations that depend on modification of data are not possible without the cooperation of an QUIC endpoint. This might be possible with the introduction of a proxy which authenticates as an endpoint. Proxy operations are not in scope for this document.

Network management is not a one-size-fits-all endeavour: practices considered necessary or even mandatory within enterprise networks with certain compliance requirements, for example, would be impermissible on other networks without those requirements. This document therefore does not make any specific recommendations as to which practices should or should not be applied; for each practice, it describes what is and is not possible with the QUIC transport protocol as defined.

2. Features of the QUIC Wire Image

In this section, we discuss those aspects of the QUIC transport protocol that have an impact on the design and operation of devices that forward QUIC packets. Here, we are concerned primarily with the unencrypted part of QUIC's wire image [WIRE-IMAGE], which we define as the information available in the packet header in each QUIC packet, and the dynamics of that information. Since QUIC is a versioned protocol, the wire image of the header format can also change from version to version. However, the field that identifies the QUIC version in some packets, and the format of the Version Negotiation Packet, are both inspectable and invariant [QUIC-INVARIANTS].

This document describes version 1 of the QUIC protocol, whose wire image is fully defined in [QUIC-TRANSPORT] and [QUIC-TLS]. Features of the wire image described herein may change in future versions of the protocol, except when specified as an invariant [QUIC-INVARIANTS], and cannot be used to identify QUIC as a protocol or to infer the behavior of future versions of QUIC.

Appendix A provides non-normative guidance on the identification of QUIC version 1 packets compared to some pre-standard versions.

2.1. QUIC Packet Header Structure

QUIC packets may have either a long header or a short header. The first bit of the QUIC header is the Header Form bit, and indicates which type of header is present. The purpose of this bit is invariant across QUIC versions.

The long header exposes more information. In version 1 of QUIC, it is used during connection establishment, including version negotiation, retry, and 0-RTT data. It contains a version number, as well as source and destination connection IDs for grouping packets belonging to the same flow. The definition and location of these fields in the QUIC long header are invariant for future versions of QUIC, although future versions of QUIC may provide additional fields in the long header [QUIC-INVARIANTS].

Short headers contain only an optional destination connection ID and the spin bit for RTT measurement. In version 1 of QUIC, they are used after connection establishment.

The following information is exposed in QUIC packet headers in all versions of QUIC:

* version number: the version number is present in the long header, and identifies the version used for that packet. During Version Negotiation (see Section 17.2.1 of [QUIC-TRANSPORT] and Section 2.8), the version number field has a special value (0x00000000) that identifies the packet as a Version Negotiation packet. QUIC version 1 uses version 0x00000001. Operators should expect to observe packets with other version numbers as a result of various Internet experiments, future standards, and greasing. All deployed versions are maintained in an IANA registry (see Section 22.2 of [QUIC-TRANSPORT]).

* source and destination connection ID: short and long packet headers carry a destination connection ID, a variable-length field that can be used to identify the connection associated with a QUIC packet, for load-balancing and NAT rebinding purposes; see Section 4.4 and Section 2.6. Long packet headers additionally carry a source connection ID. The source connection ID corresponds to the destination connection ID the source would like to have on packets sent to it, and is only present on long packet headers. On long header packets, the length of the connection IDs is also present; on short header packets, the length of the destination connection ID is implicit.

In version 1 of QUIC, the following additional information is exposed:

* "fixed bit": The second-most-significant bit of the first octet of most QUIC packets of the current version is set to 1, enabling endpoints to demultiplex with other UDP-encapsulated protocols. Even though this bit is fixed in the version 1 specification, endpoints might use an extension that varies the bit. Therefore, observers cannot reliably use it as an identifier for QUIC.

* latency spin bit: The third-most-significant bit of the first octet in the short packet header for version 1. The spin bit is set by endpoints such that tracking edge transitions can be used to passively observe end-to-end RTT. See Section 3.8.2 for further details.

* header type: The long header has a 2 bit packet type field following the Header Form and fixed bits. Header types correspond to stages of the handshake; see Section 17.2 of [QUIC-TRANSPORT] for details.

* length: The length of the remaining QUIC packet after the length field, present on long headers. This field is used to implement coalesced packets during the handshake (see Section 2.2).

* token: Initial packets may contain a token, a variable-length opaque value optionally sent from client to server, used for validating the client's address. Retry packets also contain a token, which can be used by the client in an Initial packet on a subsequent connection attempt. The length of the token is explicit in both cases.

Retry (Section 17.2.5 of [QUIC-TRANSPORT]) and Version Negotiation (Section 17.2.1 of [QUIC-TRANSPORT]) packets are not encrypted or obfuscated in any way. For other kinds of packets, version 1 of QUIC cryptographically obfuscates other information in the packet headers:

* packet number: All packets except Version Negotiation and Retry packets have an associated packet number; however, this packet number is encrypted, and therefore not of use to on-path observers. The offset of the packet number is encoded in long headers, while it is implicit (depending on destination connection ID length) in short headers. The length of the packet number is cryptographically obfuscated.

* key phase: The Key Phase bit, present in short headers, specifies the keys used to encrypt the packet to support key rotation. The Key Phase bit is cryptographically obfuscated.

2.2. Coalesced Packets

Multiple QUIC packets may be coalesced into a UDP datagram, with a datagram carrying one or more long header packets followed by zero or one short header packets. When packets are coalesced, the Length fields in the long headers are used to separate QUIC packets; see Section 12.2 of [QUIC-TRANSPORT]. The length header field is variable length, and its position in the header is also variable depending on the length of the source and destination connection ID; see Section 17.2 of [QUIC-TRANSPORT].

2.3. Use of Port Numbers

Applications that have a mapping for TCP as well as QUIC are expected to use the same port number for both services. However, as for all other IETF transports [RFC7605], there is no guarantee that a specific application will use a given registered port, or that a given port carries traffic belonging to the respective registered service, especially when application layer information is encrypted. For example, [QUIC-HTTP] specifies the use of Alt-Svc for discovery of HTTP/3 services on other ports.

Further, as QUIC has a connection ID, it is also possible to maintain multiple QUIC connections over one 5-tuple. However, if the connection ID is zero-length, all packets of the 5-tuple belong to the same QUIC connection.

2.4. The QUIC Handshake

New QUIC connections are established using a handshake, which is distinguishable on the wire and contains some information that can be passively observed.

To illustrate the information visible in the QUIC wire image during the handshake, we first show the general communication pattern visible in the UDP datagrams containing the QUIC handshake, then examine each of the datagrams in detail.

The QUIC handshake can normally be recognized on the wire through at least four datagrams we'll call "Client Initial", "Server Initial", and "Client Completion", and "Server Completion", for purposes of this illustration, as shown in Figure 1.

Packets in the handshake belong to three separate cryptographic and transport contexts ("Initial", which contains observable payload, and "Handshake" and "1-RTT", which do not). QUIC packets in separate contexts during the handshake are generally coalesced (see Section 2.2) in order to reduce the number of UDP datagrams sent during the handshake.

As shown here, the client can send 0-RTT data as soon as it has sent its Client Hello, and the server can send 1-RTT data as soon as it has sent its Server Hello.

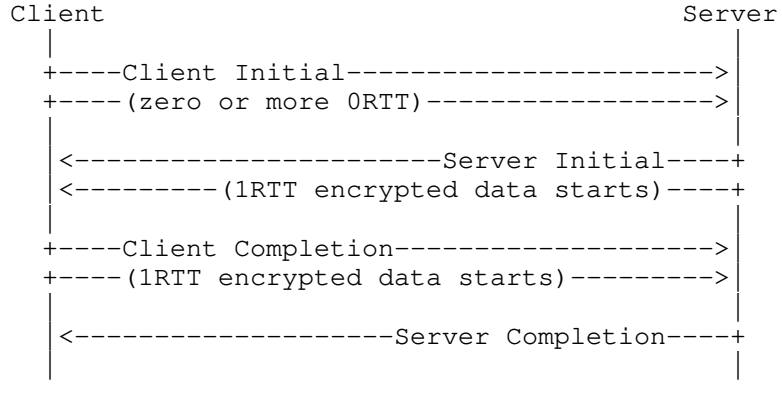


Figure 1: General communication pattern visible in the QUIC handshake

A typical handshake starts with the client sending of a Client Initial datagram as shown in Figure 2, which elicits a Server Initial datagram as shown in Figure 3 typically containing three packets: an Initial packet with the Server Initial, a Handshake packet with the rest of the server's side of the TLS handshake, and initial 1-RTT data, if present.

The Client Completion datagram contains at least one Handshake packet and some also include an Initial packet.

Datagrams that contain a Client Initial Packet (Client Initial, Server Initial, and some Client Completion) contain at least 1200 octets of UDP payload. This protects against amplification attacks and verifies that the network path meets the requirements for the minimum QUIC IP packet size; see Section 14 of [QUIC-TRANSPORT]. This is accomplished by either adding PADDING frames within the Initial packet, coalescing other packets with the Initial packet, or leaving unused payload in the UDP packet after the Initial packet. A network path needs to be able to forward at least this size of packet for QUIC to be used.

The content of Client Initial packets are encrypted using Initial Secrets, which are derived from a per-version constant and the client's destination connection ID; they are therefore observable by any on-path device that knows the per-version constant. They are therefore considered visible in this illustration. The content of QUIC Handshake packets are encrypted using keys established during the initial handshake exchange, and are therefore not visible.

Initial, Handshake, and the Short Header packets transmitted after the handshake belong to cryptographic and transport contexts. The Client Completion Figure 4 and the Server Completion Figure 5 datagrams finish these first two contexts, by sending the final acknowledgment and finishing the transmission of CRYPTO frames.

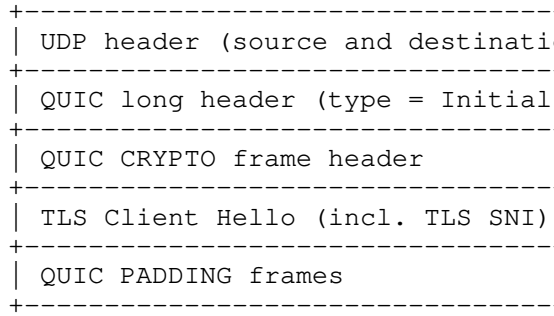


Figure 2: Typical Client Initial datagram pattern without 0-RTT

The Client Initial datagram exposes version number, source and destination connection IDs without encryption. Information in the TLS Client Hello frame, including any TLS Server Name Indication (SNI) present, is obfuscated using the Initial secret. Note that the location of PADDING is implementation-dependent, and PADDING frames might not appear in a coalesced Initial packet.

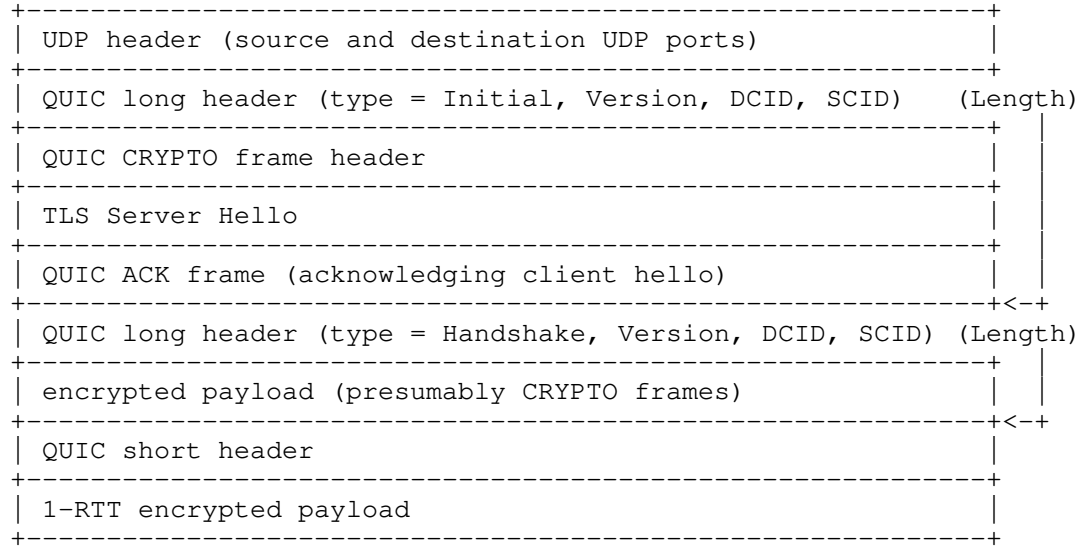


Figure 3: Typical Server Initial datagram pattern

The Server Initial datagram also exposes version number, source and destination connection IDs in the clear; information in the TLS Server Hello message is obfuscated using the Initial secret.

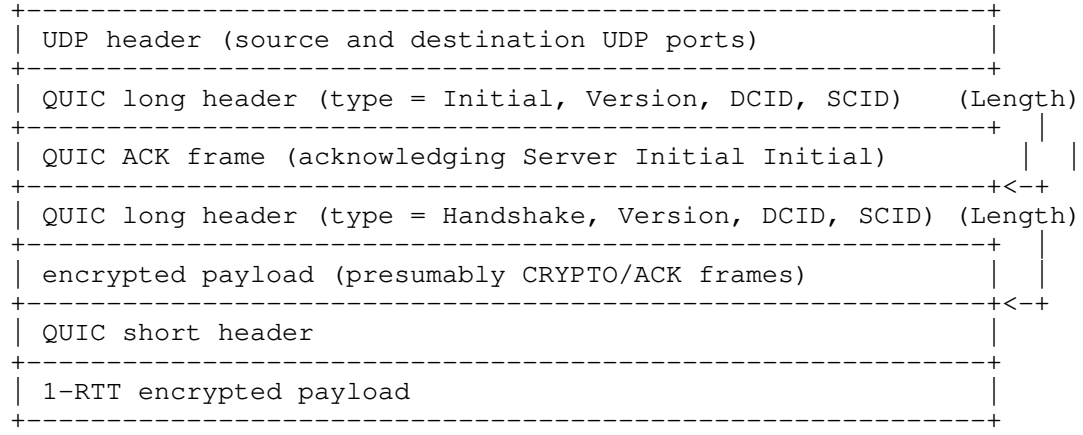


Figure 4: Typical Client Completion datagram pattern

The Client Completion datagram does not expose any additional information; however, recognizing it can be used to determine that a handshake has completed (see Section 3.2), and for three-way handshake RTT estimation as in Section 3.8.

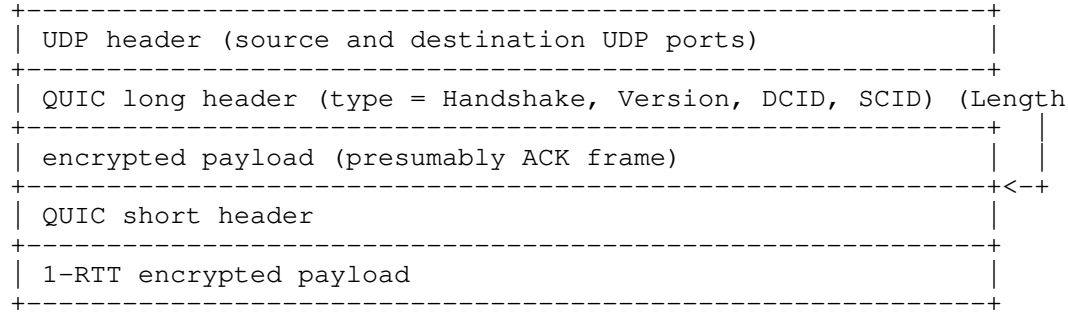


Figure 5: Typical Server Completion datagram pattern

Similar to Client Completion, Server Completion also exposes no additional information; observing it serves only to determine that the handshake has completed.

When the client uses 0-RTT connection resumption, 0-RTT data may also be seen in the Client Initial datagram, as shown in Figure 6.

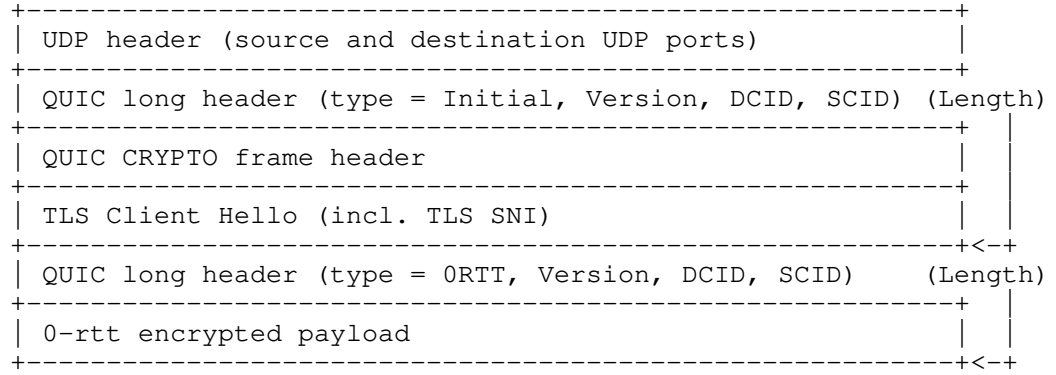


Figure 6: Typical 0-RTT Client Initial datagram pattern

In a 0-RTT Client Initial datagram, the PADDING frame is only present if necessary to increase the size of the datagram with 0RTT data to at least 1200 bytes. Additional datagrams containing only 0-RTT protected long header packets may be sent from the client to the

server after the Client Initial datagram, containing the rest of the 0-RTT data. The amount of 0-RTT protected data that can be sent in the first round is limited by the initial congestion window, typically around 10 packets (see Section 7.2 of [QUIC-RECOVERY]).

2.5. Integrity Protection of the Wire Image

As soon as the cryptographic context is established, all information in the QUIC header, including exposed information, is integrity-protected. Further, information that was exposed in packets sent before the cryptographic context was established is validated during the cryptographic handshake. Therefore, devices on path cannot alter any information or bits in QUIC packets. Such alterations would cause the integrity check to fail, which results in the receiver discarding the packet. Some parts of Initial packets could be altered by removing and re-applying the authenticated encryption without immediate discard at the receiver. However, the cryptographic handshake validates most fields and any modifications in those fields will result in connection establishment failing later on.

2.6. Connection ID and Rebinding

The connection ID in the QUIC packet headers allows association of QUIC packets using information independent of the five-tuple. This allows rebinding of a connection after one of one endpoint experienced an address change - usually the client. Further it can be used by in-network devices to ensure that related 5-tuple flows are appropriately balanced together.

Client and server negotiate connection IDs during the handshake; typically, however, only the server will request a connection ID for the lifetime of the connection. Connection IDs for either endpoint may change during the lifetime of a connection, with the new connection ID being supplied via encrypted frames (see Section 5.1 of [QUIC-TRANSPORT]). Therefore, observing a new connection ID does not necessary indicate a new connection.

[QUIC_LB] specifies algorithms for encoding the server mapping in a connection ID in order to share this information with selected on-path devices such as load balancers. Server mappings should only be exposed to selected entities. Uncontrolled exposure would allow linkage of multiple IP addresses to the same host if the server also supports migration which opens an attack vector on specific servers or pools. The best way to obscure an encoding is to appear random to any other observers, which is most rigorously achieved with encryption. As a result any attempt to infer information from specific parts of a connection ID is unlikely to be useful.

2.7. Packet Numbers

The packet number field is always present in the QUIC packet header in version 1; however, it is always encrypted. The encryption key for packet number protection on handshake packets sent before cryptographic context establishment is specific to the QUIC version, while packet number protection on subsequent packets uses secrets derived from the end-to-end cryptographic context. Packet numbers are therefore not part of the wire image that is visible to on-path observers.

2.8. Version Negotiation and Greasing

Version Negotiation packets are used by the server to indicate that a requested version from the client is not supported (see Section 6 of [QUIC-TRANSPORT]. Version Negotiation packets are not intrinsically protected, but future QUIC versions will use later encrypted messages to verify that they were authentic. Therefore any modification of this list will be detected and may cause the endpoints to terminate the connection attempt.

Also note that the list of versions in the Version Negotiation packet may contain reserved versions. This mechanism is used to avoid ossification in the implementation on the selection mechanism. Further, a client may send a Initial Client packet with a reserved version number to trigger version negotiation. In the Version Negotiation packet, the connection IDs of the Client Initial packet are reflected to provide a proof of return-routability. Therefore, changing this information will also cause the connection to fail.

QUIC is expected to evolve rapidly, so new versions, both experimental and IETF standard versions, will be deployed in the Internet more often than with traditional Internet- and transport-layer protocols. Using a particular version number to recognize valid QUIC traffic is likely to persistently miss a fraction of QUIC flows and completely fail in the near future, and is therefore not recommended. In addition, due to the speed of evolution of the protocol, devices that attempt to distinguish QUIC traffic from non-QUIC traffic for purposes of network admission control should admit all QUIC traffic regardless of version.

3. Network-Visible Information about QUIC Flows

This section addresses the different kinds of observations and inferences that can be made about QUIC flows by a passive observer in the network based on the wire image in Section 2. Here we assume a bidirectional observer (one that can see packets in both directions in the sequence in which they are carried on the wire) unless noted.

3.1. Identifying QUIC Traffic

The QUIC wire image is not specifically designed to be distinguishable from other UDP traffic.

The only application binding defined by the IETF QUIC WG is HTTP/3 [QUIC-HTTP] at the time of this writing; however, many other applications are currently being defined and deployed over QUIC, so an assumption that all QUIC traffic is HTTP/3 is not valid. HTTP/3 uses UDP port 443 by default, although URLs referring to resources available over HTTP/3 may specify alternate port numbers. Simple assumptions about whether a given flow is using QUIC based upon a UDP port number may therefore not hold; see also Section 5 of [RFC7605].

While the second-most-significant bit (0x40) of the first octet is set to 1 in most QUIC packets of the current version (see Section 2.1 and Section 17 of [QUIC-TRANSPORT]), this method of recognizing QUIC traffic is not reliable. First, it only provides one bit of information and is prone to collision with UDP-based protocols other than those considered in [RFC7983]. Second, this feature of the wire image is not invariant [QUIC-INVARIANTS] and may change in future versions of the protocol, or even be negotiated during the handshake via the use of an extension.

Even though transport parameters transmitted in the client's Initial packet are observable by the network, they cannot be modified by the network without risking connection failure. Further, the reply from the server cannot be observed, so observers on the network cannot know which parameters are actually in use.

3.1.1. Identifying Negotiated Version

An in-network observer assuming that a set of packets belongs to a QUIC flow can infer the version number in use by observing the handshake: for QUIC version 1, if the version number in the Initial packet from a client is the same as the version number in the Initial packet of the server response, that version has been accepted by both endpoints to be used for the rest of the connection.

The negotiated version cannot be identified for flows for which a handshake is not observed, such as in the case of connection migration; however, it might be possible to associate a flow with a flow for which a version has been identified; see Section 3.5.

3.1.2. First Packet Identification for Garbage Rejection

A related question is whether the first packet of a given flow on a port known to be associated with QUIC is a valid QUIC packet. This determination supports in-network filtering of garbage UDP packets (reflection attacks, random backscatter, etc.). While heuristics based on the first byte of the packet (packet type) could be used to separate valid from invalid first packet types, the deployment of such heuristics is not recommended, as bits in the first byte may have different meanings in future versions of the protocol.

3.2. Connection Confirmation

This document focuses on QUIC version 1, and this section applies only to packets belonging to QUIC version 1 flows; for purposes of on-path observation, it assumes that these packets have been identified as such through the observation of a version number exchange as described above.

Connection establishment uses Initial and Handshake packets containing a TLS handshake, and Retry packets that do not contain parts of the handshake. Connection establishment can therefore be detected using heuristics similar to those used to detect TLS over TCP. A client initiating a connection may also send data in 0-RTT packets directly after the Initial packet containing the TLS Client Hello. Since these packets may be reordered in the network, 0-RTT packets could be seen before the Initial packet.

Note that in this version of QUIC, clients send Initial packets before servers do, servers send Handshake packets before clients do, and only clients send Initial packets with tokens. Therefore, an endpoint can be identified as a client or server by an on-path observer. An attempted connection after Retry can be detected by correlating the contents of the Retry packet with the Token and the Destination Connection ID fields of the new Initial packet.

3.3. Distinguishing Acknowledgment Traffic

Some deployed in-network functions distinguish pure-acknowledgment (ACK) packets from packets carrying upper-layer data in order to attempt to enhance performance, for example by queueing ACKs differently or manipulating ACK signaling. Distinguishing ACK packets is trivial in TCP, but not supported by QUIC, since acknowledgment signaling is carried inside QUIC's encrypted payload, and ACK manipulation is impossible. Specifically, heuristics attempting to distinguish ACK-only packets from payload-carrying packets based on packet size are likely to fail, and are not recommended to use as a way to construe internals of QUIC's operation as those mechanisms can change, e.g., due to the use of extensions.

3.4. Server Name Indication (SNI)

The client's TLS ClientHello may contain a Server Name Indication (SNI) [RFC6066] extension, by which the client reveals the name of the server it intends to connect to, in order to allow the server to present a certificate based on that name. It may also contain an Application-Layer Protocol Negotiation (ALPN) [RFC7301] extension, by which the client exposes the names of application-layer protocols it supports; an observer can deduce that one of those protocols will be used if the connection continues.

Work is currently underway in the TLS working group to encrypt the contents of the ClientHello in TLS 1.3 [TLS-ECH]. This would make SNI-based application identification impossible by on-path observation for QUIC and other protocols that use TLS.

3.4.1. Extracting Server Name Indication (SNI) Information

If the ClientHello is not encrypted, it can be derived from the client's Initial packet by calculating the Initial secret to decrypt the packet payload and parsing the QUIC CRYPTO Frame containing the TLS ClientHello.

As both the derivation of the Initial secret and the structure of the Initial packet itself are version-specific, the first step is always to parse the version number (second to sixth bytes of the long header). Note that only long header packets carry the version number, so it is necessary to also check if the first bit of the QUIC packet is set to 1, indicating a long header.

Note that proprietary QUIC versions, that have been deployed before standardization, might not set the first bit in a QUIC long header packet to 1. To parse these versions, example code is provided in the appendix (see Appendix A). However, it is expected that these versions will gradually disappear over time.

When the version has been identified as QUIC version 1, the packet type needs to be verified as an Initial packet by checking that the third and fourth bits of the header are both set to 0. Then the Destination Connection ID needs to be extracted to calculate the Initial secret using the version-specific Initial salt, as described in Section 5.2 of [QUIC-TLS]. The length of the connection ID is indicated in the 6th byte of the header followed by the connection ID itself.

To determine the end of the header and find the start of the payload, the packet number length, the source connection ID length, and the token length need to be extracted. The packet number length is defined by the seventh and eight bits of the header as described in Section 17.2 of [QUIC-TRANSPORT], but is obfuscated as described in Section 5.4 of [QUIC-TLS]. The source connection ID length is specified in the byte after the destination connection ID. The token length, which follows the source connection ID, is a variable-length integer as specified in Section 16 of [QUIC-TRANSPORT].

After decryption, the client's Initial packet can be parsed to detect the CRYPTO frame that contains the TLS ClientHello, which then can be parsed similarly to TLS over TCP connections. The client's Initial packet may contain other frames, so the first bytes of each frame need to be checked to identify the frame type, and if needed skip over it. Note that the length of the frames is dependent on the frame type. In QUIC version 1, the packet is expected to contain only CRYPTO frames and optionally PADDING frames. PADDING frames, each consisting of a single zero byte, may occur before, after, or between CRYPTO frames. There might be multiple CRYPTO frames. Finally, an extension might define additional frame types which could be present.

Note that subsequent Initial packets might contain a Destination Connection ID other than the one used to generate the Initial secret. Therefore, attempts to decrypt these packets using the procedure above might fail unless the Initial secret is retained by the observer.

3.5. Flow Association

The QUIC connection ID (see Section 2.6) is designed to allow a coordinating on-path device, such as a load-balancer, to associate two flows when one of the endpoints changes address or port. This change can be due to NAT rebinding or address migration.

The connection ID must change upon intentional address change by an endpoint, and connection ID negotiation is encrypted, so it is not possible for a passive observer to link intended changes of address using the connection ID.

When one endpoint unintentionally changes its address, as is the case with NAT rebinding, an on-path observer may be able to use the connection ID to associate the flow on the new address with the flow on the old address.

A network function that attempts to use the connection ID to associate flows must be robust to the failure of this technique. Since the connection ID may change multiple times during the lifetime of a connection, packets with the same five-tuple but different connection IDs might or might not belong to the same connection. Likewise, packets with the same connection ID but different five-tuples might not belong to the same connection, either.

Connection IDs should be treated as opaque; see Section 4.4 for caveats regarding connection ID selection at servers.

3.6. Flow Teardown

QUIC does not expose the end of a connection; the only indication to on-path devices that a flow has ended is that packets are no longer observed. Stateful devices on path such as NATs and firewalls must therefore use idle timeouts to determine when to drop state for QUIC flows; see Section 4.2.

3.7. Flow Symmetry Measurement

QUIC explicitly exposes which side of a connection is a client and which side is a server during the handshake. In addition, the symmetry of a flow (whether primarily client-to-server, primarily server-to-client, or roughly bidirectional, as input to basic traffic classification techniques) can be inferred through the measurement of data rate in each direction. While QUIC traffic is protected and ACKs may be padded, padding is not required.

3.8. Round-Trip Time (RTT) Measurement

The round-trip time of QUIC flows can be inferred by observation once per flow, during the handshake, as in passive TCP measurement; this requires parsing of the QUIC packet header and recognition of the handshake, as illustrated in Section 2.4. It can also be inferred during the flow's lifetime, if the endpoints use the spin bit facility described below and in Section 17.3.1 of [QUIC-TRANSPORT].

3.8.1. Measuring Initial RTT

In the common case, the delay between the client's Initial packet (containing the TLS ClientHello) and the server's Initial packet (containing the TLS ServerHello) represents the RTT component on the path between the observer and the server. The delay between the server's first Handshake packet and the Handshake packet sent by the client represents the RTT component on the path between the observer and the client. While the client may send 0-RTT packets after the Initial packet during connection re-establishment, these can be ignored for RTT measurement purposes.

Handshake RTT can be measured by adding the client-to-observer and observer-to-server RTT components together. This measurement necessarily includes any transport- and application-layer delay (the latter mainly caused by the asymmetric crypto operations associated with the TLS handshake) at both sides.

3.8.2. Using the Spin Bit for Passive RTT Measurement

The spin bit provides a version-specific method to measure per-flow RTT from observation points on the network path throughout the duration of a connection. See Section 17.4 of [QUIC-TRANSPORT] for the definition of the spin bit in Version 1 of QUIC. Endpoint participation in spin bit signaling is optional. That is, while its location is fixed in this version of QUIC, an endpoint can unilaterally choose to not support "spinning" the bit.

Use of the spin bit for RTT measurement by devices on path is only possible when both endpoints enable it. Some endpoints may disable use of the spin bit by default, others only in specific deployment scenarios, e.g. for servers and clients where the RTT would reveal the presence of a VPN or proxy. To avoid making these connections identifiable based on the usage of the spin bit, all endpoints randomly disable "spinning" for at least one eighth of connections, even if otherwise enabled by default. An endpoint not participating in spin bit signaling for a given connection can use a fixed spin value for the duration of the connection, or can set the bit randomly on each packet sent.

When in use and a QUIC flow sends data continuously, the latency spin bit in each direction changes value once per round-trip time (RTT). An on-path observer can observe the time difference between edges (changes from 1 to 0 or 0 to 1) in the spin bit signal in a single direction to measure one sample of end-to-end RTT. This mechanism follows the principles of protocol measurability laid out in [IPIM].

Note that this measurement, as with passive RTT measurement for TCP, includes any transport protocol delay (e.g., delayed sending of acknowledgements) and/or application layer delay (e.g., waiting for a response to be generated). It therefore provides devices on path a good instantaneous estimate of the RTT as experienced by the application.

However, application-limited and flow-control-limited senders can have application and transport layer delay, respectively, that are much greater than network RTT. When the sender is application-limited and e.g. only sends small amount of periodic application traffic, where that period is longer than the RTT, measuring the spin bit provides information about the application period, not the network RTT.

Since the spin bit logic at each endpoint considers only samples from packets that advance the largest packet number, signal generation itself is resistant to reordering. However, reordering can cause problems at an observer by causing spurious edge detection and therefore inaccurate (i.e., lower) RTT estimates, if reordering occurs across a spin-bit flip in the stream.

Simple heuristics based on the observed data rate per flow or changes in the RTT series can be used to reject bad RTT samples due to lost or reordered edges in the spin signal, as well as application or flow control limitation; for example, QoF [TMA-QOF] rejects component RTTs significantly higher than RTTs over the history of the flow. These heuristics may use the handshake RTT as an initial RTT estimate for a given flow. Usually such heuristics would also detect if the spin is either constant or randomly set for a connection.

An on-path observer that can see traffic in both directions (from client to server and from server to client) can also use the spin bit to measure "upstream" and "downstream" component RTT; i.e, the component of the end-to-end RTT attributable to the paths between the observer and the server and the observer and the client, respectively. It does this by measuring the delay between a spin edge observed in the upstream direction and that observed in the downstream direction, and vice versa.

Raw RTT samples generated using these techniques can be processed in various ways to generate useful network performance metrics. A simple linear smoothing or moving minimum filter can be applied to the stream of RTT samples to get a more stable estimate of application-experienced RTT. RTT samples measured from the spin bit can also be used to generate RTT distribution information, including minimum RTT (which approximates network RTT over longer time windows) and RTT variance (which approximates jitter as seen by the application).

4. Specific Network Management Tasks

In this section, we review specific network management and measurement techniques and how QUIC's design impacts them.

4.1. Passive Network Performance Measurement and Troubleshooting

Limited RTT measurement is possible by passive observation of QUIC traffic; see Section 3.8. No passive measurement of loss is possible with the present wire image. Extremely limited observation of upstream congestion may be possible via the observation of CE markings on ECN-enabled QUIC traffic.

4.2. Stateful Treatment of QUIC Traffic

Stateful treatment of QUIC traffic (e.g., at a firewall or NAT middlebox) is possible through QUIC traffic and version identification (Section 3.1) and observation of the handshake for connection confirmation (Section 3.2). The lack of any visible end-of-flow signal (Section 3.6) means that this state must be purged either through timers or through least-recently-used eviction, depending on application requirements.

While QUIC has no clear network-visible end-of-connection signal and therefore does require timer-based state removal, the QUIC handshake indicates confirmation by both ends of a valid bidirectional transmission. As soon as the handshake completed, timers should be set long enough to also allow for short idle time during a valid transmission.

[RFC4787] requires a timeout that is not less than 2 minutes for most UDP traffic. However, in practice, timers are sometimes lower, in the range of 30 to 60 seconds. In contrast, [RFC5382] recommends a timeout of more than 2 hours for TCP, given that TCP is a connection-oriented protocol with well- defined closure semantics.

Even though QUIC has explicitly been designed tolerate NAT rebindings, decreasing the NAT timeout is not recommended, as it may negatively impact application performance or incentivize endpoints to send very frequent keep-alive packets. Instead it is recommended, even when lower timers are used for other UDP traffic, to use a timer of at least two minutes for QUIC traffic.

If state is removed too early, this could lead to black-holing of incoming packets after a short idle period. To detect this situation, a timer at the client needs to expire before a re-establishment can happen (if at all), which would lead to unnecessary long delays in an otherwise working connection.

Furthermore, not all endpoints use routing architectures where connections will survive a port or address change. So even when the client revives the connection, a NAT rebinding can cause a routing mismatch where a packet is not even delivered to the server that might support address migration. For these reasons, the limits in [RFC4787] are important to avoid black-holing of packets (and hence avoid interrupting the flow of data to the client), especially where devices are able to distinguish QUIC traffic from other UDP payloads.

The QUIC header optionally contains a connection ID which could provide additional entropy beyond the 5-tuple. The QUIC handshake needs to be observed in order to understand whether the connection ID is present and what length it has. However, connection IDs may be renegotiated after the handshake, and this renegotiation is not visible to the path. Therefore using the connection ID as a flow key field for stateful treatment of flows is not recommended as connection ID changes will cause undetectable and unrecoverable loss of state in the middle of a connection. Specially, the use of the connection ID for functions that require state to make a forwarding decison is not viable as it will break connectivity or at minimum cause long timeout-based delays before this problem is detected by the endpoints and the connection can potentially be re-established.

Use of connection IDs is specifically discouraged for NAT applications. If a NAT hits an operational limit, it is recommended to rather drop the initial packets of a flow (see also Section 4.5), which potentially triggers a fallback to TCP. Use of the connection ID to multiplex multiple connections on the same IP address/port pair is not a viable solution as it risks connectivity breakage, in case the connection ID changes.

4.3. Address Rewriting to Ensure Routing Stability

While QUIC's migration capability makes it possible for an server to survive address changes, this does not work if the routers or switches in the server infrastructure route using the address-port 4-tuple. If infrastructure routes on addresses only, NAT rebinding or address migration will cause packets to be delivered to the wrong server. [QUIC_LB] describes a way to addresses this problem by coordinating the selection and use of connection IDs between load-balancers and servers.

Applying address translation at a middlebox to maintain a stable address-port mapping for flows based on connection ID might seem like a solution to this problem. However, hiding information about the change of the IP address or port conceals important and security-relevant information from QUIC endpoints and as such would facilitate amplification attacks (see Section 9 of [QUIC-TRANSPORT]). A NAT function that hides peer address changes prevents the other end from detecting and mitigating attacks as the endpoint cannot verify connectivity to the new address using QUIC PATH_CHALLENGE and PATH_RESPONSE frames.

In addition, a change of IP address or port is also an input signal to other internal mechanisms in QUIC. When a path change is detected, path-dependent variables like congestion control parameters will be reset protecting the new path from overload.

Therefore, the use of address rewriting to ensure routing stability can open QUIC up to various attacks, as it conceals client address changes, and as such masks important signals that drive security mechanisms.

4.4. Server Cooperation with Load Balancers

In the case of networking architectures that include load balancers, the connection ID can be used as a way for the server to signal information about the desired treatment of a flow to the load balancers. Guidance on assigning connection IDs is given in [QUIC-APPLICABILITY]. [QUIC_LB] describes a system for coordinating selection and use of connection IDs between load-balancers and servers.

4.5. Filtering Behavior

[RFC4787] describes possible packet filtering behaviors that relate to NATs but is often also used is other scenarios where packet filtering is desired. Though the guidance there holds, a particularly unwise behavior is to admit a handful of UDP packets and then make a decision as to whether or not to filter it. QUIC applications are encouraged to fail over to TCP if early packets do not arrive at their destination [I-D.ietf-quic-applicability], as QUIC is based on UDP and there are known blocks of UDP traffic (see Section 4.6). Admitting a few packets allows the QUIC endpoint to determine that the path accepts QUIC. Sudden drops afterwards will result in slow and costly timeouts before abandoning the connection.

4.6. UDP Blocking or Throttling

Today, UDP is the most prevalent DDoS vector, since it is easy for compromised non-admin applications to send a flood of large UDP packets (while with TCP the attacker gets throttled by the congestion controller) or to craft reflection and amplification attacks. Some networks therefore block UDP traffic. With increased deployment of QUIC, there is also an increased need to allow UDP traffic on ports used for QUIC. However, if UDP is generally enabled on these ports, UDP flood attacks may also use the same ports. One possible response to this threat is to throttle UDP traffic on the network, allocating a fixed portion of the network capacity to UDP and blocking UDP datagrams over that cap. As the portion of QUIC traffic compared to TCP is also expected to increase over time, using such a limit is not recommended but if done, limits might need to be adapted dynamically.

Further, if UDP traffic is desired to be throttled, it is recommended to block individual QUIC flows entirely rather than dropping packets randomly. When the handshake is blocked, QUIC-capable applications may failover to TCP However, blocking a random fraction of QUIC packets across 4-tuples will allow many QUIC handshakes to complete, preventing a TCP failover, but the connections will suffer from severe packet loss (see also Section 4.5). Therefore UDP throttling should be realized by per-flow policing as opposed to per-packet policing. Note that this per-flow policing should be stateless to avoid problems with stateful treatment of QUIC flows (see Section 4.2), for example blocking a portion of the space of values of a hash function over the addresses and ports in the UDP datagram. While QUIC endpoints are often able to survive address changes, e.g. by NAT rebindings, blocking a portion of the traffic based on 5-tuple hashing increases the risk of black-holing an active connection when the address changes.

4.7. DDoS Detection and Mitigation

On-path observation of the transport headers of packets can be used for various security functions. For example, Denial of Service (DOS) and Distributed DOS (DDOS) attacks against the infrastructure or against an endpoint can be detected and mitigated by characterising anomalous traffic. Other uses include support for security audits (e.g., verifying the compliance with ciphersuites); client and application fingerprinting for inventory; and to provide alerts for network intrusion detection and other next generation firewall functions.

Current practices in detection and mitigation of DDoS attacks generally involve classification of incoming traffic (as packets, flows, or some other aggregate) into "good" (productive) and "bad" (DDoS) traffic, and then differential treatment of this traffic to forward only good traffic. This operation is often done in a separate specialized mitigation environment through which all traffic is filtered; a generalized architecture for separation of concerns in mitigation is given in [DOTS-ARCH].

Efficient classification of this DDoS traffic in the mitigation environment is key to the success of this approach. Limited first-packet garbage detection as in Section 3.1.2 and stateful tracking of QUIC traffic as in Section 4.2 above may be useful during classification.

Note that the use of a connection ID to support connection migration renders 5-tuple based filtering insufficient to detect active flows and requires more state to be maintained by DDoS defense systems if support of migration of QUIC flows is desired. For the common case of NAT rebinding, where the client's address changes without the client's intent or knowledge, DDoS defense systems can detect a change in the client's endpoint address by linking flows based on the server's connection IDs. However, QUIC's linkability resistance ensures that a deliberate connection migration is accompanied by a change in the connection ID. In this case, the connection ID can not be used to distinguish valid, active traffic from new attack traffic.

It is also possible for endpoints to directly support security functions such as DoS classification and mitigation. Endpoints can cooperate with an in-network device directly by e.g. sharing information about connection IDs.

Another potential method could use an on-path network device that relies on pattern inferences in the traffic and heuristics or machine learning instead of processing observed header information.

However, it is questionable whether connection migrations must be supported during a DDoS attack. While unintended migration without a connection ID change can be more easily supported, it might be acceptable to not support migrations of active QUIC connections that are not visible to the network functions performing the DDoS detection. As soon as the connection blocking is detected by the client, the client may be able to rely on the fast resumption mechanism provided by QUIC. When clients migrate to a new path, they should be prepared for the migration to fail and attempt to reconnect quickly.

Beyond in-network DDoS protection mechanisms, TCP syncookies [RFC4937] are a well-established method of mitigating some kinds of TCP DDoS attacks. QUIC Retry packets are the functional analogue to syncookies, forcing clients to prove possession of their IP address before committing server state. However, there are safeguards in QUIC against unsolicited injection of these packets by intermediaries who do not have consent of the end server. See [QUIC_LB] for standard ways for intermediaries to send Retry packets on behalf of consenting servers.

4.8. Quality of Service handling and ECMP

It is expected that any QoS handling in the network, e.g. based on use of DiffServ Code Points (DSCPs) [RFC2475] as well as Equal-Cost Multi-Path (ECMP) routing, is applied on a per flow-basis (and not per-packet) and as such that all packets belonging to the same QUIC connection get uniform treatment. Using ECMP to distribute packets from a single flow across multiple network paths or any other non-uniform treatment of packets belong to the same connection could result in variations in order, delivery rate, and drop rate. As feedback about loss or delay of each packet is used as input to the congestion controller, these variations could adversely affect performance.

Depending of the loss recovery mechanism implemented, QUIC may be more tolerant of packet re-ordering than traditional TCP traffic (see Section 2.7). However, it cannot be known by the network which exact recovery mechanism is used and therefore reordering tolerance should be considered as unknown.

4.9. Handling ICMP Messages

Datagram Packetization Layer PMTU Discovery (PLPMTUD) can be used by QUIC to probe for the supported PMTU. PLPMTUD optionally uses ICMP messages (e.g., IPv6 Packet Too Big messages). Given known attacks with the use of ICMP messages, the use of PLPMTUD in QUIC has been designed to safely use but not rely on receiving ICMP feedback (see Section 14.2.1. of [QUIC-TRANSPORT]).

Networks are recommended to forward these ICMP messages and retain as much of the original packet as possible without exceeding the minimum MTU for the IP version when generating ICMP messages as recommended in [RFC1812] and [RFC4443].

4.10. Guiding Path MTU

Some networks support 1500-byte packets, but can only do so by fragmenting at a lower layer before traversing a smaller MTU segment, and then reassembling. This is permissible even when the IP layer is IPv6 or IPv4 with the DF bit set, because it occurs below the IP layer. However, this process can add to compute and memory costs, leading to a bottleneck that limits network capacity. In such networks this generates a desire to influence a majority of senders to use smaller packets, so that the limited reassembly capacity is not exceeded.

For TCP, MSS clamping (Section 3.2 of [RFC4459]) is often used to change the sender's maximum TCP segment size, but QUIC requires a different approach. Section 14 of [QUIC-TRANSPORT] advises senders to probe larger sizes using Datagram Packetization Layer PMTU Discovery ([DPLPMTUD]) or Path Maximum Transmission Unit Discovery (PMTUD: [RFC1191] and [RFC8201]). This mechanism will encourage senders to approach the maximum size, which could cause fragmentation with a network segment that they may not be aware of.

If path performance is limited when sending larger packets, an on-path device should support a maximum packet size for a specific transport flow and then consistently drop all packets that exceed the configured size when the inner IPv4 packet has DF set, or IPv6 is used. Endpoints can cache PMTU information between IP flows, in the IP-layer cache, so short-term consistency between the PMTU for flows can help avoid an endpoint using a PMTU that is inefficient.

Networks with configurations that would lead to fragmentation of large packets should drop such packets rather than fragmenting them. Network operators who plan to implement a more selective policy may start by focussing on QUIC. QUIC flows cannot always be easily distinguished from other UDP traffic, but we assume at least some

portion of QUIC traffic can be identified (see Section 3.1). For QUIC endpoints using DPLPMTUD it is recommended for the path to drop a packet larger than the supported size. A QUIC probe packet is used to discover the PMTU. If lost, this does not impact the flow of QUIC data.

IPv4 routers generate an ICMP message when a packet is dropped because the link MTU was exceeded. [RFC8504] specifies how an IPv6 node generates an ICMPv6 Packet Too Big message (PTB) in this case. PMTUD relies upon an endpoint receiving such PTB messages [RFC8201], whereas DPLPMTUD does not reply upon these messages, but still can optionally use these to improve performance Section 4.6 of [DPLPMTUD].

Since a network cannot know in advance which discovery method a QUIC endpoint is using, it should always send a PTB message in addition to dropping the oversized packet. A generated PTB message should be compliant with the validation requirements of Section 14.2.1 of [QUIC-TRANSPORT], otherwise it will be ignored by DPLPMTUD. This will likely provide the right signal for the endpoint to keep the packet size small and thereby avoid network fragmentation for that flow entirely.

5. IANA Considerations

This document has no actions for IANA.

6. Security Considerations

QUIC is an encrypted and authenticated transport. That means, once the cryptographic handshake is complete, QUIC endpoints discard most packets that are not authenticated, greatly limiting the ability of an attacker to interfere with existing connections.

However, some information is still observerable, as supporting manageability of QUIC traffic inherently involves tradeoffs with the confidentiality of QUIC's control information; this entire document is therefore security-relevant.

More security considerations for QUIC are discussed in [QUIC-TRANSPORT] and [QUIC-TLS], generally considering active or passive attackers in the network as well as attacks on specific QUIC mechanism.

Version Negotiation packets do not contain any mechanism to prevent version downgrade attacks. However, future versions of QUIC that use Version Negotiation packets are require to define a mechanism that is robust against version downgrade attacks. Therefore a network node should not attempt to impact version selection, as version downgrade may result in connection failure.

7. Contributors

The following people have contributed text to sections of this document:

* Dan Druta
* Martin Duke
* Igor Lubashev
* David Schinazi
* Gorry Fairhurst
* Chris Box

8. Acknowledgments

Thanks to Thomas Fossati, Jana Iygengar, Marcus Ihlar for their early reviews and feedback. Special thanks also to Martin Thomson and Martin Duke for their detailed reviews and input. And thanks to Sean Turner, Mike Bishop, Ian Swett, and Nick Banks for their last call reviews.

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.

9. References

9.1. Normative References

[QUIC-TLS] Thomson, M. and S. Turner, "Using TLS to Secure QUIC", Work in Progress, Internet-Draft, draft-ietf-quic-tls-34, 14 January 2021, <https://tools.ietf.org/html/draft-ietf-quic-tls-34>.

[QUIC-TRANSPORT]
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", Work in Progress, Internet-Draft, draft-ietf-quic-transport-34, 14 January 2021, <https://tools.ietf.org/html/draft-ietf-quic-transport-34>.

9.2. Informative References

[DOTS-ARCH]
Mortensen, A., Reddy, T., Andreasen, F., Teague, N., and R. Compton, "DDoS Open Threat Signaling (DOTS) Architecture", Work in Progress, Internet-Draft, draft-ietf-dots-architecture-18, 6 March 2020, <https://tools.ietf.org/html/draft-ietf-dots-architecture-18>.

[DPLPMTUD] Fairhurst, G., Jones, T., Tüxen, M., Rüngeler, I., and T. Völker, "Packetization Layer Path MTU Discovery for Datagram Transports", RFC 8899, DOI 10.17487/RFC8899, September 2020, <https://www.rfc-editor.org/rfc/rfc8899>.

[I-D.ietf-quic-applicability]
Kuehlewind, M. and B. Trammell, "Applicability of the QUIC Transport Protocol", Work in Progress, Internet-Draft, draft-ietf-quic-applicability-11, 21 April 2021, <https://tools.ietf.org/html/draft-ietf-quic-applicability-11>.

[IPIM] Allman, M., Beverly, R., and B. Trammell, "In-Protocol Internet Measurement (arXiv preprint 1612.02902)", 9 December 2016, <https://arxiv.org/abs/1612.02902>.

[QUIC-APPLICABILITY]
Kuehlewind, M. and B. Trammell, "Applicability of the QUIC Transport Protocol", Work in Progress, Internet-Draft, draft-ietf-quic-applicability-11, 21 April 2021, <https://tools.ietf.org/html/draft-ietf-quic-applicability-11>.

[QUIC-HTTP]
Bishop, M., "Hypertext Transfer Protocol Version 3 (HTTP/3)", Work in Progress, Internet-Draft, draft-ietf-quic-http-34, 2 February 2021, <https://tools.ietf.org/html/draft-ietf-quic-http-34>.

[QUIC-INVARIANTS]
Thomson, M., "Version-Independent Properties of QUIC", Work in Progress, Internet-Draft, draft-ietf-quic-invariants-13, 14 January 2021, <https://tools.ietf.org/html/draft-ietf-quic-invariants-13>.

[QUIC-RECOVERY]
Iyengar, J. and I. Swett, "QUIC Loss Detection and Congestion Control", Work in Progress, Internet-Draft, draft-ietf-quic-recovery-34, 14 January 2021, <https://tools.ietf.org/html/draft-ietf-quic-recovery-34>.

[QUIC_LB] Duke, M. and N. Banks, "QUIC-LB: Generating Routable QUIC Connection IDs", Work in Progress, Internet-Draft, draft-ietf-quic-load-balancers-06, 4 February 2021, <https://tools.ietf.org/html/draft-ietf-quic-load-balancers-06>.

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

[RFC1812] Baker, F., Ed., "Requirements for IP Version 4 Routers", RFC 1812, DOI 10.17487/RFC1812, June 1995, <https://www.rfc-editor.org/rfc/rfc1812>.

[RFC2475] Blake, S., Black, D., Carlson, M., Davies, E., Wang, Z., and W. Weiss, "An Architecture for Differentiated Services", RFC 2475, DOI 10.17487/RFC2475, December 1998, <https://www.rfc-editor.org/rfc/rfc2475>.

[RFC4443] Conta, A., Deering, S., and M. Gupta, Ed., "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification", STD 89, RFC 4443, DOI 10.17487/RFC4443, March 2006, <https://www.rfc-editor.org/rfc/rfc4443>.

[RFC4459] Savola, P., "MTU and Fragmentation Issues with In-the-Network Tunneling", RFC 4459, DOI 10.17487/RFC4459, April 2006, <https://www.rfc-editor.org/rfc/rfc4459>.

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

[RFC4937] Arberg, P. and V. Mammoliti, "IANA Considerations for PPP over Ethernet (PPPoE)", RFC 4937, DOI 10.17487/RFC4937, June 2007, <https://www.rfc-editor.org/rfc/rfc4937>.

[RFC5382] Guha, S., Ed., Biswas, K., Ford, B., Sivakumar, S., and P. Srisuresh, "NAT Behavioral Requirements for TCP", BCP 142, RFC 5382, DOI 10.17487/RFC5382, October 2008, <https://www.rfc-editor.org/rfc/rfc5382>.

[RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <https://www.rfc-editor.org/rfc/rfc6066>.

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

[RFC7605] Touch, J., "Recommendations on Using Assigned Transport Port Numbers", BCP 165, RFC 7605, DOI 10.17487/RFC7605, August 2015, <https://www.rfc-editor.org/rfc/rfc7605>.

[RFC7983] Petit-Huguenin, M. and G. Salgueiro, "Multiplexing Scheme Updates for Secure Real-time Transport Protocol (SRTP) Extension for Datagram Transport Layer Security (DTLS)", RFC 7983, DOI 10.17487/RFC7983, September 2016, <https://www.rfc-editor.org/rfc/rfc7983>.

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

[RFC8504] Chown, T., Loughney, J., and T. Winters, "IPv6 Node Requirements", BCP 220, RFC 8504, DOI 10.17487/RFC8504, January 2019, <https://www.rfc-editor.org/rfc/rfc8504>.

[TLS-ECH] Rescorla, E., Oku, K., Sullivan, N., and C. A. Wood, "TLS Encrypted Client Hello", Work in Progress, Internet-Draft, draft-ietf-tls-esni-10, 8 March 2021, <https://tools.ietf.org/html/draft-ietf-tls-esni-10>.

[TMA-QOF] Trammell, B., Gugelmann, D., and N. Brownlee, "Inline Data Integrity Signals for Passive Measurement (in Proc. TMA 2014)", April 2014.

[WIRE-IMAGE]
Trammell, B. and M. Kuehlewind, "The Wire Image of a Network Protocol", RFC 8546, DOI 10.17487/RFC8546, April 2019, <https://www.rfc-editor.org/rfc/rfc8546>.

Appendix A. Distinguishing IETF QUIC and Google QUIC Versions

This section contains algorithms that allows parsing versions from both Google QUIC and IETF QUIC. These mechanisms will become irrelevant when IETF QUIC is fully deployed and Google QUIC is deprecated.

Note that other than this appendix, nothing in this document applies to Google QUIC. And the purpose of this appendix is merely to distinguish IETF QUIC from any versions of Google QUIC.

This appendix uses the following conventions: * array[i] - one byte at index i of array * array[i:j] - subset of array starting with index i (inclusive) up to j-1 (inclusive) * array[i:] - subset of array starting with index i (inclusive) up to the end of the array

Conceptually, a Google QUIC version is an opaque 32bit field. When we refer to a version with four printable characters, we use its ASCII representation: for example, Q050 refers to {'Q', '0', '5', '0'} which is equal to {0x51, 0x30, 0x35, 0x30}. Otherwise, we use its hexadecimal representation: for example, 0xff00001d refers to {0xff, 0x00, 0x00, 0x1d}.

QUIC versions that start with 'Q' or 'T' followed by three digits are Google QUIC versions. Versions up to and including 43 are documented by <https://docs.google.com/document/d/1WJvyZflAO2pq77yOLbp9NsGjC1CHetAXV8I0fQe-B_U/preview>. Versions Q046, Q050, T050, and T051 are not fully documented, but this appendix should contain enough information to allow parsing Client Hellos for those versions.

To extract the version number itself, one needs to look at the first byte of the QUIC packet, in other words the first byte of the UDP payload.

```
first_byte = packet[0]
first_byte_bit1 = ((first_byte & 0x80) != 0)
first_byte_bit2 = ((first_byte & 0x40) != 0)
first_byte_bit3 = ((first_byte & 0x20) != 0)
first_byte_bit4 = ((first_byte & 0x10) != 0)
first_byte_bit5 = ((first_byte & 0x08) != 0)
first_byte_bit6 = ((first_byte & 0x04) != 0)
first_byte_bit7 = ((first_byte & 0x02) != 0)
first_byte_bit8 = ((first_byte & 0x01) != 0)
if (first_byte_bit1) {
  version = packet[1:5]
} else if (first_byte_bit5 && !first_byte_bit2) {
  if (!first_byte_bit8) {
    abort("Packet without version")
  }
  if (first_byte_bit5) {
    version = packet[9:13]
  } else {
    version = packet[5:9]
  }
} else {
  abort("Packet without version")
}
```

A.1. Extracting the CRYPTO frame

```
counter = 0
while (payload[counter] == 0) {
  counter += 1
}
first_nonzero_payload_byte = payload[counter]
fnz_payload_byte_bit3 = ((first_nonzero_payload_byte & 0x20) != 0)

if (first_nonzero_payload_byte != 0x06) {
  abort("Unexpected frame")
}
if (payload[counter+1] != 0x00) {
  abort("Unexpected crypto stream offset")
}
counter += 2
if ((payload[counter] & 0xc0) == 0) {
  crypto_data_length = payload[counter]
  counter += 1
} else {
  crypto_data_length = payload[counter:counter+2]
  counter += 2
}
crypto_data = payload[counter:counter+crypto_data_length]
ParseTLS(crypto_data)
```

Authors' Addresses

Mirja Kuehlewind
Ericsson

Email: mirja.kuehlewind@ericsson.com

Brian Trammell
Google Switzerland GmbH
Gustav-Gull-Platz 1
CH- 8004 Zurich
Switzerland

Email: ietf@trammell.ch

QUIC Working Group
Internet-Draft
Intended status: Informational
Expires: 9 August 2021

D. Schinazi
Google LLC
E. Rescorla
Mozilla

5 February 2021

Compatible Version Negotiation for QUIC
draft-ietf-quic-version-negotiation-03

Abstract

QUIC does not provide a complete version negotiation mechanism but instead only provides a way for the server to indicate that the version the client offered is unacceptable. This document describes a version negotiation mechanism that allows a client and server to select a mutually supported version. Optionally, if the original and negotiated version share a compatible first flight format, the negotiation can take place without incurring an extra round trip.

Discussion of this work is encouraged to happen on the QUIC IETF mailing list quic@ietf.org (mailto:quic@ietf.org) or on the GitHub repository which contains the draft: https://github.com/quicwg/version-negotiation/ (https://github.com/quicwg/version-negotiation/).

Discussion Venues

This note is to be removed before publishing as an RFC.

Source for this draft and an issue tracker can be found at https://github.com/quicwg/version-negotiation.

Status of This Memo

This Internet-Draft will expire on 9 August 2021.

Table of Contents

1. Introduction

The version-invariant properties of QUIC [INV] define a version negotiation (VN) packet but do not specify how an endpoint reacts when it receives one. QUIC version 1 [QUIC] allows the server to use a VN packet to indicate that the version the client offered is unacceptable, but doesn't allow the client to safely make use of that information to create a new connection with a mutually supported

version. With proper safety mechanisms in place, the VN packet can be part of a mechanism to allow two QUIC implementations to negotiate between two totally disjoint versions of QUIC, at the cost of an extra round trip. However, it is beneficial to avoid that cost whenever possible, especially given that most incremental versions are broadly similar to the the previous version.

This specification describes a simple version negotiation mechanism which optionally leverages similarities between versions and can negotiate between the set of "compatible" versions in a single round trip.

Discussion of this work is encouraged to happen on the QUIC IETF mailing list quic@ietf.org (mailto:quic@ietf.org) or on the GitHub repository which contains the draft: https://github.com/quicwg/version-negotiation/ (https://github.com/quicwg/version-negotiation/).

1.1. Conventions and Definitions

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

2. Compatible Versions

If A and B are two distinct versions of QUIC, A is said to be "compatible" with B if it is possible to take a first flight of packets from version A and convert it into a first flight of packets from version B. As an example, if versions A and B are absolutely equal in their wire image and behavior during the handshake but differ after the handshake, then A is compatible with B and B is compatible with A.

Version compatibility is not bijective: it is possible for version A to be compatible with version B and for B not to be compatible with A. This could happen for example if version B is a strict superset of version A.

Note that version compatibility does not mean that every single possible instance of a first flight will succeed in conversion to the other version. A first flight using version A is said to be "compatible" with version B if two conditions are met: first that version A is compatible with version B, and second that the conversion of this first flight to version B is well-defined. For example, if version B is equal to A in all aspects except it

introduced a new frame in its first flight that version A cannot parse or even ignore, then B could still be compatible with A as conversions would succeed for connections where that frame is not used. In this example, first flights using version B that carry this new frame would not be compatible with version A.

When a new version of QUIC is defined, it is assumed to not be compatible with any other version unless otherwise specified. Similarly, no other version is compatible with the new version unless otherwise specified. Implementations MUST NOT assume compatibility between versions unless explicitly specified.

Note that both endpoints might disagree on whether two versions are compatible or not. For example, two versions could have been defined concurrently and then specified as compatible in a third document much later - in that scenario one endpoint might be aware of the compatibility document while the other may not.

3. Version Negotiation Mechanism

This document specifies two means of performing version negotiation: one "incompatible" which requires a round trip and is applicable to all versions, and one "compatible" that allows saving the round trip but only applies when the versions are compatible.

The client initiates a QUIC connection by sending a first flight of QUIC packets with a long header to the server [INV]. We'll refer to the version of those packets as the "original version". The client's first flight includes handshake version information (see Section 4) which will be used to optionally enable compatible version negotation (see Section 3.3), and to prevent version downgrade attacks (see Section 5).

Upon receiving this first flight, the server verifies whether it knows how to parse first flights from the original version. If it does not, then it starts incompatible version negotiation, see Section 3.2. If the server can parse the first flight, it can either establish the connection using the original version, or it MAY attempt compatible version negotiation, see Section 3.3.

Note that it is possible for a server to have the ability to parse the first flight of a given version without fully supporting it, in the sense that it implements enough of the version's specification to parse first flight packets but not enough to fully establish a connection using that version.

3.1. Connections and Version Negotiation

QUIC connections are shared state between a client and a server [INV]. The compatible version negotiation mechanism defined in this document (see Section 3.3) operates inside of a QUIC connection; i.e., the packets with the original version are part of the same connection as the packets with the negotiated version. On the other hand, the incompatible version negotiation mechanism, which leverages QUIC Version Negotiation packets (see Section 3.2) conceptually operates across two QUIC connections: one before the Version Negotiation packet, and a distinct connection after.

3.2. Incompatible Version Negotiation

The server starts incompatible version negotiation by sending a VN packet, listing all the versions that it does support.

Upon receiving the VN packet, the client will search for a version it supports in the list provided by the server. If it doesn't find one, it aborts the connection attempt. Otherwise, it selects a mutually supported version and sends a new first flight with that version - we refer to this version as the "negotiated version".

The new first flight will allow the endpoints to establish a connection using the negotiated version. The handshake of the negotiated version will exchange handshake version information (see Section 4) required to ensure that VN was genuine, i.e. that no attacker injected packets in order to influence the VN process, see Section 5.

3.3. Compatible Version Negotiation

When the server can parse the client's first flight using the original version, it can extract the client's handshake version information (see Section 4). This contains the list of versions that the client thinks its first flight is compatible with.

If the server supports one of the client's compatible versions, and the server also believes that the original version is compatible with this version, then the server converts the client's first flight to that version and replies to the client as if it had received the converted first flight. The version used by the server in its reply is refered to as the "negotiated version". The server MUST NOT reply with a version that is not present in the client's compatible versions, unless it is the original version.

If the server does not find a compatible version, it will use the original version if it supports it, and if it doesn't then the server will perform incompatible version negotiation instead, see Section 3.2.

For the duration of the compatible version negotiation process, clients MUST use the same 5-tuple (source and destination IP addresses and UDP port numbers). During that time, clients MUST also use the same Destination Connection ID, except if the server explicitly instructs the client to use a different Destination Connection ID (for example, a QUIC version 1 server can accomplish this by sending an INITIAL packet with a Source Connection ID that differed from the client's Destination Connection ID). This allows load balancers to ensure that packets for a given connection are routed to the same server.

4. Handshake Version Information

During the handshake, endpoints will exchange handshake version information, which is a blob of data that is defined below. In QUIC version 1, the handshake version information is transmitted using a new transport parameter, "version_negotiation". The contents of handshake version information depend on whether the client or server is sending it, and are shown below (using the notation from the "Notational Conventions" section of [QUIC]):

```
Client Handshake Version Information {
  Currently Attempted Version (32),
  Previously Attempted Version (32),
  Received Negotiation Version Count (i),
  Received Negotiation Version (32) ...,
  Compatible Version Count (i),
  Compatible Version (32) ...,
}
```

Figure 1: Client Handshake Version Information

The content of each field is described below:

Currently Attempted Version: The version that the client is attempting to use. This field MUST be equal to the value of the Version field in the long header that carries this data.

Previously Attempted Version: If the client is sending this first

flight in response to a Version Negotiation packet, this field contains the version that the client used in the previous first flight that triggered the version negotiation packet. If the client did not receive a Version Negotiation packet, this field SHALL be all-zeroes.

Received Negotiation Version Count: A variable-length integer specifying the number of Received Negotiation Version fields following it. If the client is sending this first flight in response to a Version Negotiation packet, the subsequent versions SHALL include all the versions from that Version Negotiation packet in order, even if they are not supported by the client (even if the versions are reserved). If the client has not received a Version Negotiation packet on this connection, this field SHALL be 0.

Compatible Version Count: A variable-length integer specifying the number of Compatible Version fields following it. The client lists all versions that this first flight is compatible with in the subsequent Compatible Version fields, ordered by descending preference. Note that the version in the Currently Attempted Version field MUST be included in the Compatible Version list to allow the client to communicate the currently attempted version's preference. Note that this preference is only advisory, servers MAY choose to use their own preference instead.

```
Server Handshake Version Information {
  Negotiated Version (32),
  Supported Version Count (i),
  Supported Version (32) ...,
}
```

Figure 2: Server Handshake Version Information

The content of each field is described below:

Negotiated Version: The version that the server chose to use for the connection. This field MUST be equal to the value of the Version field in the long header that carries this data.

Supported Version Count: A variable-length integer specifying the number of Supported Version fields following it. The server encodes all versions it supports in the subsequent list, ordered by descending preference. Note that the version in the Negotiated Version field MUST be included in the Supported Version list to allow the server to communicate the negotiated version's preference. Note that this preference is only advisory, clients MAY choose to use their own preference instead.

Clients MAY include versions following the pattern "0x?a?a?a?a" in their "Compatible Version" list, and the server in their "Supported Version" list. Those versions are reserved to exercise version negotiation (see the Versions section of [QUIC]), and MUST be ignored when parsing these fields. On the other hand, the "Received Negotiation Version" list MUST be identical to the received Version Negotiation packet, so clients MUST NOT add or remove reserved version from that list.

5. Version Downgrade Prevention

Clients MUST ignore any received Version Negotiation packets that contain the version that they initially attempted. Once a client has reacted to a Version Negotiation packet, it MUST drop all subsequent Version Negotiation packets on that connection.

Servers MUST validate that the client's "Currently Attempted Version" matches the version in the long header that carried the handshake version information. Similarly, clients MUST validate that the server's "Negotiated Version" matches the long header version. If an endpoint's validation fails, it MUST close the connection. If the connection was using QUIC version 1, it MUST be closed with a transport error of type "VERSION_NEGOTIATION_ERROR".

When a server parses the client's handshake version information, if the "Received Negotiation Version Count" is not zero, the server MUST validate that it could have sent the Version Negotation packet described by the client in response to a first flight of version "Previously Attempted Version". In particular, the server MUST ensure that there are no versions that it supports that are absent from the Received Negotiation Versions, and that the ordering matches the server's preference. If this validation fails, the server MUST close the connection. If the connection was using QUIC version 1, it MUST be closed with a transport error of type "VERSION_NEGOTIATION_ERROR". This mitigates an attacker's ability to forge Version Negotiation packets to force a version downgrade.

If a server operator is progressively deploying a new QUIC version throughout its fleet, it MAY perform a two-step process where it first progressively adds support for the new version, but without enforcing its presence in "Received Negotiation Versions". Once all servers have been upgraded, the second step is to start enforcing that the new version is present in "Received Negotiation Versions". This opens connections to version downgrades during the upgrade window, since those could be due to clients communicating with both upgraded and non-upgraded servers.

If an endpoint receives its peer's Handshake Version Information and fails to parse it (for example, if it is too short), then the endpoint MUST close the connection. If the connection was using QUIC version 1, it MUST be closed with a transport error of type "TRANSPORT_PARAMETER_ERROR".

6. Supported Versions

The server's "Supported Version" list allows it to communicate the full list of versions it supports to the client. In the case where clients initially attempt connections with the oldest version they support, this allows them to be notified of more recent versions the server supports. If the client notices that the server supports a version that is more preferable that the one initially attempted by default, the client SHOULD cache that information and attempt the preferred version in subsequent connections.

7. Client Choice of Original Version

The client's first flight SHOULD be sent using the version that the server is most likely to support (in the absence of other information, this will often be the oldest version the client supports).

8. Interaction with Retry

QUIC version 1 features retry packets, which the server can send to validate the client's IP address before parsing the client's first flight. This impacts compatible version negotiation because a server who wishes to send a retry packet before parsing the client's first flight won't have parsed the client's handshake version information yet. If a future document wishes to define compatibility between two versions that support retry, that document MUST specify how version negotiation (both compatible and incompatible) interacts with retry during a handshake that requires both. For example, that could be accomplished by having the server send a retry packet first and validating the client's IP address before starting version negotiation and deciding whether to use compatible version negotiation on that connection (in that scenario the retry packet would be sent using the original version).

9. Interaction with 0-RTT

QUIC version 1 allows sending data from the client to the server during the handshake, by using 0-RTT packets. If a future document wishes to define compatibility between two versions that support 0-RTT, that document MUST address the scenario where there are 0-RTT packets in the client's first flight. For example, this could be accomplished by defining which transformations are applied to 0-RTT packets. Alternatively, that document could specify that compatible version negotiation causes 0-RTT data to be rejected by the server.

10. Considerations for Future Versions

In order to facilitate the deployment of future versions of QUIC, designers of future versions SHOULD attempt to design their new version such that commonly deployed versions are compatible with it. For example, a successor to QUIC version 1 may wish to design its transport parameters in a way that does not preclude compatibility. Additionally, frames in QUIC version 1 do not use a self-describing encoding, so unrecognized frame types cannot be parsed or ignored (see the Extension Frames section of [QUIC]); this means that new versions that wish to be very similar to QUIC version 1 and compatible with it should avoid introducing new frames in initial packets.

11. Security Considerations

The security of this version negotiation mechanism relies on the authenticity of the handshake version information exchanged during the handshake. In QUIC version 1, transport parameters are authenticated ensuring the security of this mechanism. Negotiation between compatible versions will have the security of the weakest common version.

The requirement that versions not be assumed compatible mitigates the possibility of cross-protocol attacks, but more analysis is still needed here.

The presence of the "Attempted Version" and "Negotiated Version" fields mitigates an attacker's ability to forge packets by altering the version.

12. IANA Considerations

12.1. QUIC Transport Parameter

If this document is approved, IANA shall assign the following entry in the QUIC Transport Parameter Registry:

Value	Parameter Name	Reference
0x73DB	version_negotiation	This document

12.2. QUIC Transport Error Code

If this document is approved, IANA shall assign the following entry in the QUIC Transport Error Codes Registry:

Value	Parameter Name	Reference
0x53F8	VERSION_NEGOTIATION_ERROR	This document

13. Normative References

[INV] Thomson, M., "Version-Independent Properties of QUIC", Work in Progress, Internet-Draft, draft-ietf-quic-invariants-13, 14 January 2021, <http://www.ietf.org/internet-drafts/draft-ietf-quic-invariants-13.txt>.

[QUIC] Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", Work in Progress, Internet-Draft, draft-ietf-quic-transport-34, 14 January 2021, <http://www.ietf.org/internet-drafts/draft-ietf-quic-transport-34.txt>.

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

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

Acknowledgments

The authors would like to thank Martin Thomson, Mike Bishop, Nick Banks, Ryan Hamilton, and Roberto Peon for their input and contributions.

Authors' Addresses

David Schinazi
Google LLC
1600 Amphitheatre Parkway
Mountain View, California 94043,
United States of America

Email: dschinazi.ietf@gmail.com

Eric Rescorla
Mozilla

Email: ekr@rtfm.com

QUIC
Internet-Draft
Intended status: Standards Track
Expires: 6 May 2021

J. Iyengar
Fastly
I. Swett
Google

2 November 2020

Sender Control of Acknowledgement Delays in QUIC
draft-iyengar-quic-delayed-ack-02

Abstract

This document describes a QUIC extension for an endpoint to control its peer's delaying of acknowledgements.

Note to Readers

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

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

Status of This Memo

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

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

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

This Internet-Draft will expire on 6 May 2021.

Copyright Notice

Table of Contents

1. Introduction

This document describes a QUIC extension for an endpoint to control its peer's delaying of acknowledgements.

1.1. Terms and Definitions

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

In the rest of this document, "sender" refers to a QUIC data sender

(and acknowledgement receiver). Similarly, "receiver" refers to a QUIC data receiver (and acknowledgement sender).

An "acknowledgement packet" refers to a QUIC packet that contains only an ACK frame.

This document uses terms, definitions, and notational conventions described in Section 1.2 and Section 1.3 of [QUIC-TRANSPORT].

2. Motivation

A receiver acknowledges received packets, but it can delay sending these acknowledgements. The delaying of acknowledgements can impact connection throughput, loss detection and congestion controller performance at a data sender, and CPU utilization at both a data sender and a data receiver.

Reducing the frequency of acknowledgement packets can improve connection and endpoint performance in the following ways:

* Sending UDP packets can be noticeably CPU intensive on some platforms. Reducing the number of packets that only contain acknowledgements can therefore reduce the amount of CPU consumed at a data receiver. Experience shows that this cost reduction can be significant for high bandwidth connections.

* Similarly, receiving and processing UDP packets can also be CPU intensive, and reducing acknowledgement frequency reduces this cost at a data sender.

* Severely asymmetric link technologies, such as DOCSIS, LTE, and satellite links, connection throughput in the data direction becomes constrained when the reverse bandwidth is filled by acknowledgment packets. When traversing such links, reducing the number of acknowledgments allows connection throughput to scale much further.

As discussed in Section 8 however, there are undesirable consequences to congestion control and loss recovery if a receiver uniltaerally reduces the acknowledgment frequency. A sender’s constraints on the acknowledgement frequency need to be taken into account to maximize congestion controller and loss recovery performance.

[QUIC-TRANSPORT] currently specifies a simple delayed acknowledgement mechanism that a receiver can use: send an acknowledgement for every other packet, and for every packet when reordering is observed. This simple mechanism does not allow a sender to signal its constraints. This extension provides a mechanism to solve this problem.

3. Negotiating Extension Use

Endpoints advertise their support of the extension described in this document by sending the following transport parameter (Section 7.2 of [QUIC-TRANSPORT]):

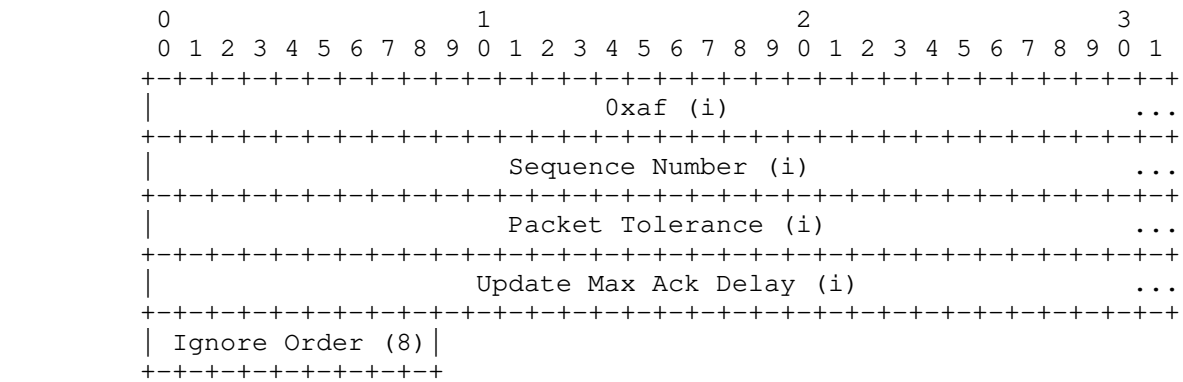
min_ack_delay (0xff02de1a): A variable-length integer representing the minimum amount of time in microseconds by which the endpoint can delay an acknowledgement. Values of 2^24 or greater are invalid, and receipt of these values MUST be treated as a connection error of type TRANSPORT_PARAMETER_ERROR.

An endpoint's min_ack_delay MUST NOT be greater than the its max_ack_delay. Endpoints that support this extension MUST treat receipt of a min_ack_delay that is greater than the received max_ack_delay as a connection error of type TRANSPORT_PARAMETER_ERROR. Note that while the endpoint's max_ack_delay transport parameter is in milliseconds (Section 18.2 of [QUIC-TRANSPORT]), min_ack_delay is specified in microseconds.

This Transport Parameter is encoded as per Section 18 of [QUIC-TRANSPORT].

4. ACK_FREQUENCY Frame

Delaying acknowledgements as much as possible reduces both work done by the endpoints and network load. An endpoint's loss detection and congestion control mechanisms however need to be tolerant of this delay at the peer. An endpoint signals its tolerance to its peer using an ACK_FREQUENCY frame, shown below:



Following the common frame format described in Section 12.4 of [QUIC-TRANSPORT], ACK_FREQUENCY frames have a type of 0xaf, and contain the following fields:

Sequence Number: A variable-length integer representing the sequence number assigned to the ACK_FREQUENCY frame by the sender to allow receivers to ignore obsolete frames, see Section 5.

Packet Tolerance: A variable-length integer representing the maximum number of ack-eliciting packets after which the receiver sends an acknowledgement. A value of 1 will result in an acknowledgement being sent for every ack-eliciting packet received. A value of 0 is invalid. Receipt of an invalid value MUST be treated as a connection error of type FRAME_ENCODING_ERROR.

Update Max Ack Delay: A variable-length integer representing an update to the peer's "max_ack_delay" transport parameter (Section 18.2 of [QUIC-TRANSPORT]). The value of this field is in microseconds. Any value smaller than the "min_ack_delay" advertised by this endpoint is invalid. Receipt of an invalid value MUST be treated as a connection error of type PROTOCOL_VIOLATION.

Ignore Order: An 8-bit field representing a boolean truth value. This field MUST have the value 0x00 (representing "false") or 0x01 (representing "true"). This field can be set to "true" by an endpoint that does not wish to receive an immediate acknowledgement when the peer observes reordering (Section 6.1). Receipt of any other value MUST be treated as a connection error of type FRAME_ENCODING_ERROR.

ACK_FREQUENCY frames are ack-eliciting. However, their loss does not require retransmission if an ACK_FREQUENCY frame with a larger Sequence Number value has been sent.

An endpoint MAY send ACK_FREQUENCY frames multiple times during a connection and with different values.

An endpoint will have committed a "max_ack_delay" value to the peer, which specifies the maximum amount of time by which the endpoint will delay sending acknowledgments. When the endpoint receives an ACK_FREQUENCY frame, it MUST update this maximum time to the value proposed by the peer in the Update Max Ack Delay field.

5. Multiple ACK_FREQUENCY Frames

An endpoint can send multiple ACK_FREQUENCY frames, and each one of them can have different values in all fields. An endpoint MUST use a sequence number of 0 for the first ACK_FREQUENCY frame it constructs and sends, and a strictly increasing value thereafter.

An endpoint MUST allow reordered ACK_FREQUENCY frames to be received and processed, see Section 13.3 of [QUIC-TRANSPORT].

On the first received ACK_FREQUENCY frame in a connection, an endpoint MUST immediately record all values from the frame. The sequence number of the frame is recorded as the largest seen sequence number. The new Packet Tolerance and Update Max Ack Delay values MUST be immediately used for delaying acknowledgements; see Section 6.

On a subsequently received ACK_FREQUENCY frame, the endpoint MUST check if this frame is more recent than any previous ones, as follows:

* If the frame's sequence number is not greater than the largest one seen so far, the endpoint MUST ignore this frame.

* If the frame's sequence number is greater than the largest one seen so far, the endpoint MUST immediately replace old recorded state with values received in this frame. The endpoint MUST start using the new values immediately for delaying acknowledgements; see Section 6. The endpoint MUST also replace the recorded sequence number.

6. Sending Acknowledgments

Prior to receiving an ACK_FREQUENCY frame, endpoints send acknowledgements as specified in Section 13.2.1 of [QUIC-TRANSPORT].

On receiving an ACK_FREQUENCY frame and updating its recorded "max_ack_delay" and "Packet Tolerance" values (Section 5), the endpoint MUST send an acknowledgement when one of the following conditions are met:

* Since the last acknowledgement was sent, the number of received ack-eliciting packets is greater than or equal to the recorded "Packet Tolerance".

* Since the last acknowledgement was sent, "max_ack_delay" amount of time has passed.

Section 6.1, Section 6.2, and Section 6.3 describe exceptions to this strategy.

An endpoint is expected to bundle acknowledgements when possible. Every time an acknowledgement is sent, bundled or otherwise, all counters and timers related to delaying of acknowledgments are reset.

6.1. Response to Reordering

As specified in Section 13.3.1 of [QUIC-TRANSPORT], endpoints are expected to send an acknowledgement immediately on receiving a reordered ack-eliciting packet. This extension modifies this behavior.

If the endpoint has not yet received an ACK_FREQUENCY frame, or if the most recent frame received from the peer has an "Ignore Order" value of "false" (0x00), the endpoint MUST immediately acknowledge any subsequent packets that are received out of order.

If the most recent ACK_FREQUENCY frame received from the peer has an "Ignore Order" value of "true" (0x01), the endpoint does not make this exception. That is, the endpoint MUST NOT send an immediate acknowledgement in response to packets received out of order, and instead continues to use the peer's "Packet Tolerance" and "max_ack_delay" thresholds for sending acknowledgements.

6.2. Expediting Congestion Signals

As specified in Section 13.3.1 of [QUIC-TRANSPORT], an endpoint SHOULD immediately acknowledge packets marked with the ECN Congestion Experienced (CE) codepoint in the IP header. Doing so reduces the peer's response time to congestion events.

6.3. Batch Processing of Packets

For performance reasons, an endpoint can receive incoming packets from the underlying platform in a batch of multiple packets. This batch can contain enough packets to cause multiple acknowledgements to be sent.

To avoid sending multiple acknowledgements in rapid succession, an endpoint MAY process all packets in a batch before determining whether a threshold has been met and an acknowledgement is to be sent in response.

7. Computation of Probe Timeout Period

On sending an update to the peer's "max_ack_delay", an endpoint can use this new value in later computations of its Probe Timeout (PTO) period; see Section 5.2.1 of [QUIC-RECOVERY]. The endpoint MUST however wait until the ACK_FREQUENCY frame that carries this new value is acknowledged by the peer.

Until the frame is acknowledged, the endpoint MUST use the greater of the current "max_ack_delay" and the value that is in flight when

computing the PTO period. Doing so avoids spurious PTOs that can be caused by an update that increases the peer's "max_ack_delay".

While it is expected that endpoints will have only one ACK_FREQUENCY frame in flight at any given time, this extension does not prohibit having more than one in flight. Generally, when using "max_ack_delay" for PTO computations, endpoints MUST use the maximum of the current value and all those in flight.

8. Implementation Considerations

There are tradeoffs inherent in a sender sending an ACK_FREQUENCY frame to the receiver. As such it is recommended that implementers experiment with different strategies and find those which best suit their applications and congestion controllers. There are, however, noteworthy considerations when devising strategies for sending ACK_FREQUENCY frames.

8.1. Loss Detection

A sender relies on receipt of acknowledgements to determine the amount of data in flight and to detect losses, e.g. when packets experience reordering, see [QUIC-RECOVERY]. Consequently, how often a receiver sends acknowledgments determines how long it takes for losses to be detected at the sender.

8.2. New Connections

Many congestion control algorithms have a startup mechanism during the beginning phases of a connection. It is typical that in this period the congestion controller will quickly increase the amount of data in the network until it is signalled to stop. While the mechanism used to achieve this increase varies, acknowledgments by the peer are generally critical during this phase to drive the congestion controller's machinery. A sender can send ACK_FREQUENCY frames while its congestion controller is in this state, ensuring that the receiver will send acknowledgments at a rate which is optimal for the the sender's congestion controller.

8.3. Window-based Congestion Controllers

Congestion controllers that are purely window-based and strictly adherent to packet conservation, such as the one defined in [QUIC-RECOVERY], rely on receipt of acknowledgments to move the congestion window forward and send additional data into the network. Such controllers will suffer degraded performance if acknowledgments are delayed excessively. Similarly, if these controllers rely on the timing of peer acknowledgments (an "ACK clock"), delaying

acknowledgments will cause undesirable bursts of data into the network.

9. Security Considerations

TBD.

10. IANA Considerations

TBD.

11. Normative References

[QUIC-RECOVERY]
Iyengar, J., Ed. and I. Swett, Ed., "QUIC Loss Detection and Congestion Control", Work in Progress, Internet-Draft, draft-ietf-quic-recovery-latest, <https://tools.ietf.org/html/draft-ietf-quic-recovery-latest>.

[QUIC-TRANSPORT]
Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", Work in Progress, Internet-Draft, draft-ietf-quic-transport-latest, <https://tools.ietf.org/html/draft-ietf-quic-transport-latest>.

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

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

Appendix A. Change Log

RFC Editor's Note: Please remove this section prior to publication of a final version of this document.

Acknowledgments

The following people directly contributed key ideas that shaped this draft: Bob Briscoe, Kazuho Oku, Marten Seemann.

Authors' Addresses

Jana Iyengar
Fastly

Email: jri.ietf@gmail.com

Ian Swett
Google

Email: ian.swett@google.com

quic
Internet-Draft
Intended status: Standards Track
Expires: 22 May 2021

M. Thomson
Mozilla

18 November 2020

Greasing the QUIC Bit
draft-thomson-quic-bit-grease-01

Abstract

This document describes a method for negotiating the ability to send an arbitrary value for the second-to-most significant bit in QUIC packets.

Discussion Venues

This note is to be removed before publishing as an RFC.

Discussion of this document takes place on the QUIC Working Group mailing list (quic@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/quic/ (https://mailarchive.ietf.org/arch/browse/quic/).

Source for this draft and an issue tracker can be found at https://github.com/martinthomson/quic-bit-grease (https://github.com/martinthomson/quic-bit-grease).

Table of Contents

1. Introduction

QUIC [QUIC] intentionally describes a very narrow set of fields that are visible to entities other than endpoints. Beyond those characteristics that are defined as invariant [QUIC-INVARIANTS], very little about the "wire image" [RFC8546] of QUIC is visible.

The second-to-most significant bit of the first byte in every QUIC packet is defined as having a fixed value in QUIC version 1 [QUIC]. The purpose of having a fixed value is to allow intermediaries and endpoints to efficiently distinguish between QUIC and other protocols; see [DEMUX] for a description of a scheme that QUIC can integrate with as a result. As this bit effectively identifies a packet as QUIC, it is sometimes referred to as the "QUIC Bit".

Where endpoints and the intermediaries that support them do not depend on the QUIC Bit having a fixed value, sending the same value in every packet is more of liability than an asset. If systems come to depend on a fixed value, then it might become infeasible to define a version of QUIC that attributes semantics to this bit.

In order to safeguard future use of this bit, this document defines a QUIC transport parameter that indicates that an endpoint is willing to receive QUIC packets containing any value for this bit. By sending different values for this bit, the hope is that the value will remain available for future use [USE-IT].

2. Conventions and Definitions

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

This document uses terms and notational conventions from [QUIC].

3. The Grease QUIC Bit Transport Parameter

The grease_quic_bit transport parameter (0x2ab2) can be sent by both client and server. The transport parameter is sent with an empty value; an endpoint that understands this transport parameter MUST treat receipt of a non-empty value as a connection error of type TRANSPORT_PARAMETER_ERROR.

Advertising the grease_quic_bit transport parameter indicates that packets sent to this endpoint MAY set a value of 0 for the QUIC Bit. The QUIC Bit is defined as the second-to-most significant bit of the first byte of QUIC packets (that is, the value 0x40).

A server MUST respect the value it previously provided for the grease_quic_bit transport parameter if it accepts 0-RTT. A client MAY forget the value. In all other cases, only the presence or absence of the transport parameter in the current handshake is used to determine what values can be sent in the QUIC Bit.

3.1. Clearing the QUIC Bit

Endpoints that receive the grease_quic_bit transport parameter from a peer MAY set the QUIC Bit to any value in packets they sent to that peer. Endpoints SHOULD set the QUIC Bit to an unpredictable value unless another extension assigns specific meaning to the value of the bit. All packets sent after receiving and processing transport parameters are affected, including Retry, Initial, and Handshake packets.

A client MAY also clear the QUIC Bit in Initial packets that are sent to establish a new connection. A client can only clear the QUIC Bit if the packet includes a token provided by the server in a NEW_TOKEN frame on a connection where the server also included the grease_quic_bit transport parameter. To allow for changes in server configuration, clients SHOULD set the QUIC Bit if the token was provided more than 7 days prior.

3.2. Using the QUIC Bit

The purpose of this extension is to allow for the use of the QUIC Bit by later extensions.

Extensions to QUIC that define semantics for the QUIC Bit can be negotiated at the same time as the grease_quic_bit transport parameter. In this case, a recipient needs to be able to distinguish a randomized value from a value carrying information according to the extension. Extensions that use the QUIC Bit MUST negotiate their use prior to acting on any semantic. Endpoints MAY send a signal prior to this negotiation completing, but any value carried by the bit cannot be used until it is clear that the peer is using the extension.

For example, an extension might define a transport parameter that is sent in addition to the grease_quic_bit transport parameter. Though the value of the QUIC Bit in packets received by a peer might be set according to rules defined by the extension, they might also be randomized according to the definition of the grease_quic_bit extension. Receiving the transport parameter for the extension could be used to confirm that a peer supports the semantic defined in the extension. To avoid acting on a randomized signal, the extension can require that endpoints set the QUIC Bit according to the rules of the extension, but defer acting on the information conveyed until the transport parameter for the extension is received.

Extensions that define semantics for the QUIC Bit can be negotiated without using the grease_quic_bit transport parameter.

4. Security Considerations

This document introduces no new security considerations for endpoints or entities that can rely on endpoint cooperation. However, this change makes the task of identifying QUIC more difficult without cooperation of endpoints. This sometimes works counter to the security goals of network operators who rely on network classification to identify threats.

5. IANA Considerations

This document registers the grease_quic_bit transport parameter in the "QUIC Transport Parameters" registry established in Section 22.2 of [QUIC]. The following fields are registered:

Value: 0x2ab2

Parameter Name: grease_quic_bit

Status: Permanent

Specification: This document.

Date: Date of registration.

Contact: QUIC Working Group (quic@ietf.org)

Change Controller: IETF (iesg@ietf.org)

Notes: (none)

6. References

6.1. Normative References

[QUIC] Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", Work in Progress, Internet-Draft, draft-ietf-quic-transport-32, 20 October 2020, <http://www.ietf.org/internet-drafts/draft-ietf-quic-transport-32.txt>.

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

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

6.2. Informative References

[DEMUX] Petit-Huguenin, M. and G. Salgueiro, "Multiplexing Scheme Updates for Secure Real-time Transport Protocol (SRTP) Extension for Datagram Transport Layer Security (DTLS)", RFC 7983, DOI 10.17487/RFC7983, September 2016, <https://www.rfc-editor.org/info/rfc7983>.

[QUIC-INVARIANTS] Thomson, M., "Version-Independent Properties of QUIC", Work in Progress, Internet-Draft, draft-ietf-quic-invariants-11, 24 September 2020, <http://www.ietf.org/internet-drafts/draft-ietf-quic-invariants-11.txt>.

[RFC8546] Trammell, B. and M. Kuehlewind, "The Wire Image of a Network Protocol", RFC 8546, DOI 10.17487/RFC8546, April 2019, <https://www.rfc-editor.org/info/rfc8546>.

[USE-IT] Thomson, M., "Long-term Viability of Protocol Extension Mechanisms", Work in Progress, Internet-Draft, draft-iab-use-it-or-lose-it-00, 7 August 2019, <http://www.ietf.org/internet-drafts/draft-iab-use-it-or-lose-it-00.txt>.

Acknowledgments

TODO

Author's Address

Martin Thomson
Mozilla

Email: mt@lowentropy.net