

Network Working Group
Internet-Draft
Intended status: Informational
Expires: 8 October 2022

M. Kuehlewind
Ericsson
B. Trammell
Google
6 April 2022

Applicability of the QUIC Transport Protocol
draft-ietf-quic-applicability-16

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 8 October 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction	2
2. The Necessity of Fallback	3
3. Zero RTT	4
3.1. Replay Attacks	4
3.2. Session resumption versus Keep-alive	5
4. Use of Streams	7
4.1. Stream versus Flow Multiplexing	8
4.2. Prioritization	9
4.3. Ordered and Reliable Delivery	9
4.4. Flow Control Deadlocks	10
4.5. Stream Limit Commitments	11
5. Packetization and Latency	12
6. Error Handling	13
7. Acknowledgment Efficiency	14
8. Port Selection and Application Endpoint Discovery	14
8.1. Source Port Selection	15
9. Connection Migration	16
10. Connection Termination	16
11. Information Exposure and the Connection ID	17
11.1. Server-Generated Connection ID	18
11.2. Mitigating Timing Linkability with Connection ID Migration	18
11.3. Using Server Retry for Redirection	19
12. Quality of Service (QoS) and DSCP	19
13. Use of Versions and Cryptographic Handshake	20
14. Enabling New Versions	20
15. Unreliable Datagram Service over QUIC	21
16. IANA Considerations	22
17. Security Considerations	22
18. Contributors	22
19. Acknowledgments	23
20. References	23
20.1. Normative References	23
20.2. Informative References	23
Authors' Addresses	27

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.

Measurement studies have shown between three [Trammell16] and five [Swett16] percent of networks block all UDP traffic, though there is little evidence of other forms of systematic disadvantage to UDP traffic compared to TCP [Edeline16]. This blocking implies that all applications running on top of QUIC must 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. This is particularly relevant for QUIC as it addresses the implications of fallback among multiple protocols.

Specifically, fallback to insecure protocols or to weaker versions of secure protocols needs to be avoided. In general, a 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 1.3, which is used in QUIC version 1, 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 multiple 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 no 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 mobility. 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 that has been 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 it is safe to use 0-RTT and when the client is the restarting peer.

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. 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, a given packet exposes no information about which stream(s) are carried within the packet. 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. When a QUIC endpoint uses fully reliable streams for transmission, prioritization of retransmissions will be beneficial in most cases, 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 Section 4 of [QUIC] 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 and communicated a cumulative stream limit might 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 peer 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 GOAWAY 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 sent 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. These are signaled using STOP_SENDING or RESET_STREAM frames, which contain only an error code.

7. Acknowledgment Efficiency

QUIC version 1 without extensions uses an acknowledgment strategy adopted from TCP Section 13.2 of [QUIC]). That is, it recommends every other packet is acknowledged. However, generating and processing QUIC acknowledgments consumes resources at a sender and receiver. Acknowledgments also incur forwarding costs and contribute to link utilization, which can impact performance over some types of network. Applications might be able to improve overall performance by using alternative strategies that reduce the rate of acknowledgments.

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.

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.

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 [RFC7838] for an HTTP origin to advertise the availability of an equivalent HTTP/3 endpoint on a certain UDP port by using the "h3" Application-Layer Protocol Negotiation (ALPN) [RFC7301] token.

ALPN 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 are required to register an ALPN token for use in the TLS handshake.

As QUIC version 1 deferred defining a complete version negotiation mechanism, HTTP/3 requires QUIC version 1 and defines the ALPN token ("h3") to only apply to that version. So far no single approach has been selected for managing the use of different QUIC versions, neither in HTTP/3 nor in general. Application protocols that use QUIC need to consider how the protocol will manage different QUIC versions. Decisions for those protocols might be informed by choices made by other protocols, like HTTP/3.

8.1. Source Port Selection

Some UDP protocols are vulnerable to reflection attacks, where an attacker is able to direct traffic to a third party as a denial of service. For example, these source ports are associated with applications known to be vulnerable to reflection attacks, often due to server misconfiguration:

- * port 53 - DNS [RFC1034]
- * port 123 - NTP [RFC5905]
- * port 1900 - SSDP [SSDP]
- * port 5353 - mDNS [RFC6762]
- * port 11211 - memcached

Services might block source ports associated with protocols known to be vulnerable to reflection attacks, to avoid the overhead of processing large numbers of packets. However, this practice has negative effects on clients: not only does it require establishment of a new connection, but in some instances, might cause the client to avoid using QUIC for that service for a period of time, downgrading to a non-UDP protocol (see Section 2).

As a result, client implementations are encouraged to avoid using source ports associated with protocols known to be vulnerable to reflection attacks. Note that the list above is not exhaustive; other source ports might be considered reflection vectors as well.

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. 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 active migration is supported. If a connection is intentionally migrated to new path, a new connection ID is used to minimize linkability by network observers. The other QUIC endpoint uses the connection ID to link different addresses to the same connection and entity if a non-zero-length connection ID is provided.

The base specification of QUIC version 1 only supports the use of a single network path at a time, which enables failover use cases. Path validation is required so that endpoints validate paths before use 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. Probing packets cannot carry application data but likely 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 decision 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. For more information on manageability of QUIC see also [I-D.ietf-quic-manageability]. 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 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 Retry packet such that the load balancer will redirect the next 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, as defined in [QUIC], has a single congestion controller and recovery handler. This design 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}, that have the same DiffServ Code Point (DSCP) [RFC2475] will receive similar network treatment since feedback about loss or delay of each packet is used as input to the congestion controller. Therefore, packets belonging to the same connection should use a single DSCP. Section 5.1 of [RFC7657] provides a discussion of DiffServ interactions with datagram transport protocols [RFC7657] (in this respect the interactions with QUIC resemble those of SCTP).

When multiplexing multiple flows over a single QUIC connection, the selected DSCP value should be the one associated with the highest priority requested for all multiplexed flows.

If differential network treatment is desired, e.g., by the use of different DSCPs, 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 overhead and, more importantly, competing congestion control.

As in other uses of DiffServ, when a packet enters a network segment that does not support the DSCP value, this could result in the connection not receiving the network treatment it expects. The DSCP value in this packet could also be remarked as the packet travels along the network path, changing the requested treatment.

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 negotiation 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 to 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 an 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.

Further, migration to a new address exposes a linkage between client addresses to the server and may expose this linkage also to the path if the connection ID cannot be changed or flows can otherwise be correlated. When migration is supported, this needs to be considered with respect to user privacy.

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 significant text to and/or feedback on this document:

- * Gorrry Fairhurst
- * Ian Swett

- * Igor Lubashev
- * Lucas Pardue
- * Mike Bishop
- * Mark Nottingham
- * Martin Duke
- * Martin Thomson
- * Sean Turner
- * Tommy Pauly

19. Acknowledgments

Special thanks to last-call reviewers Chris Lonvick and Ines Robles.

This work was 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., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <<https://www.rfc-editor.org/rfc/rfc9000>>.
- [QUIC-INVARIANTS] Thomson, M., "Version-Independent Properties of QUIC", RFC 8999, DOI 10.17487/RFC8999, May 2021, <<https://www.rfc-editor.org/rfc/rfc8999>>.
- [QUIC-TLS] Thomson, M., Ed. and S. Turner, Ed., "Using TLS to Secure QUIC", RFC 9001, DOI 10.17487/RFC9001, May 2021, <<https://www.rfc-editor.org/rfc/rfc9001>>.

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-12, 17 January 2022, <<https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-priority-12>>.

[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-07, 5 April 2022, <<https://datatracker.ietf.org/doc/html/draft-ietf-quic-version-negotiation-07>>.

[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-10, 4 February 2022, <<https://datatracker.ietf.org/doc/html/draft-ietf-quic-datagram-10>>.

[I-D.ietf-quic-manageability]

Kuehlewind, M. and B. Trammell, "Manageability of the QUIC Transport Protocol", Work in Progress, Internet-Draft, draft-ietf-quic-manageability-15, 7 March 2022, <<https://datatracker.ietf.org/doc/html/draft-ietf-quic-manageability-15>>.

[I-D.ietf-taps-arch]

Pauly, T., Trammell, B., Brunstrom, A., Fairhurst, G., and C. Perkins, "An Architecture for Transport Services", Work

in Progress, Internet-Draft, draft-ietf-taps-arch-12, 3 January 2022, <<https://datatracker.ietf.org/doc/html/draft-ietf-taps-arch-12>>.

[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://datatracker.ietf.org/doc/html/draft-ietf-quic-http-34>>.

[QUIC-LB]

Duke, M., Banks, N., and C. Huitema, "QUIC-LB: Generating Routable QUIC Connection IDs", Work in Progress, Internet-Draft, draft-ietf-quic-load-balancers-13, 28 March 2022, <<https://datatracker.ietf.org/doc/html/draft-ietf-quic-load-balancers-13>>.

[RFC1034]

Mockapetris, P., "Domain names - concepts and facilities", STD 13, RFC 1034, DOI 10.17487/RFC1034, November 1987, <<https://www.rfc-editor.org/rfc/rfc1034>>.

[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>>.

[RFC5905]

Mills, D., Martin, J., Ed., Burbank, J., and W. Kasch, "Network Time Protocol Version 4: Protocol and Algorithms Specification", RFC 5905, DOI 10.17487/RFC5905, June 2010, <<https://www.rfc-editor.org/rfc/rfc5905>>.

- [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>>.
- [RFC6762] Cheshire, S. and M. Krochmal, "Multicast DNS", RFC 6762, DOI 10.17487/RFC6762, February 2013, <<https://www.rfc-editor.org/rfc/rfc6762>>.
- [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>>.
- [RFC7657] Black, D., Ed. and P. Jones, "Differentiated Services (Diffserv) and Real-Time Communication", RFC 7657, DOI 10.17487/RFC7657, November 2015, <<https://www.rfc-editor.org/rfc/rfc7657>>.
- [RFC7838] Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", RFC 7838, DOI 10.17487/RFC7838, April 2016, <<https://www.rfc-editor.org/rfc/rfc7838>>.
- [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>>.
- [SSDP] Donoho, A., Roe, B., Bodlaender, M., Gildred, J., Messer, A., Kim, Y., Fairman, B., and J. Tourzan, "UPnP Device Architecture 2.0", 17 April 2020, <<https://openconnectivity.org/upnp-specs/UPnP-arch-DeviceArchitecture-v2.0-20200417.pdf>>.
- [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>>.
- [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>>.

[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

quic
Internet-Draft
Intended status: Standards Track
Expires: 14 May 2022

M. Thomson
Mozilla
10 November 2021

Greasing the QUIC Bit
draft-ietf-quic-bit-grease-02

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/quicwg/quic-bit-grease> (<https://github.com/quicwg/quic-bit-grease>).

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 14 May 2022.

Copyright Notice

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

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
2. Conventions and Definitions	3
3. The Grease QUIC Bit Transport Parameter	3
3.1. Clearing the QUIC Bit	3
3.2. Using the QUIC Bit	4
4. Security Considerations	4
5. IANA Considerations	5
6. References	5
6.1. Normative References	5
6.2. Informative References	5
Author's Address	6

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 send 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 prior to receiving transport parameters from the server. A client can only clear the QUIC Bit if such packets include 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 as specified in this document. Including both extensions allows for the QUIC Bit to be greased even if the alternative use is not supported.

Receiving a transport parameter for an extension that uses the QUIC Bit 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., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <<https://www.rfc-editor.org/rfc/rfc9000>>.
- [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/rfc/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/rfc/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/rfc/rfc7983>>.

[QUIC-INVARIANTS]

Thomson, M., "Version-Independent Properties of QUIC",
RFC 8999, DOI 10.17487/RFC8999, May 2021,
<<https://www.rfc-editor.org/rfc/rfc8999>>.

[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/rfc/rfc8546>>.

[USE-IT] Thomson, M. and T. Pauly, "Long-term Viability of Protocol
Extension Mechanisms", Work in Progress, Internet-Draft,
draft-iab-use-it-or-lose-it-04, 12 October 2021,
<<https://datatracker.ietf.org/doc/html/draft-iab-use-it-or-lose-it-04>>.

Author's Address

Martin Thomson
Mozilla

Email: mt@lowentropy.net

QUIC
Internet-Draft
Intended status: Standards Track
Expires: 8 August 2022

T. Pauly
E. Kinnear
Apple Inc.
D. Schinazi
Google LLC
4 February 2022

An Unreliable Datagram Extension to QUIC
draft-ietf-quic-datagram-10

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 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 (<mailto:quic@ietf.org>), which is archived at <https://mailarchive.ietf.org/arch/browse/quic/>.

Source for this draft and an issue tracker can be found at <https://github.com/quicwg/datagram>.

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 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction	2
1.1. Specification of Requirements	3
2. Motivation	3
3. Transport Parameter	4
4. Datagram Frame Types	5
5. Behavior and Usage	6
5.1. Multiplexing Datagrams	6
5.2. Acknowledgement Handling	7
5.3. Flow Control	7
5.4. Congestion Control	8
6. Security Considerations	8
7. IANA Considerations	8
7.1. QUIC Transport Parameter	8
7.2. QUIC Frame Types	9
8. Acknowledgments	9
9. References	9
9.1. Normative References	9
9.2. Informative References	10
Authors' Addresses	10

1. Introduction

The QUIC Transport Protocol [RFC9000] provides a secure, multiplexed connection for transmitting reliable streams of application data. QUIC uses various frame types to transmit data within packets, and each frame type defines whether the data it contains will be retransmitted. Streams of reliable application data are sent using STREAM frames.

Some applications, particularly those that need to transmit real-time data, prefer to transmit data unreliably. In the past, these applications have built directly upon UDP [RFC0768] as a transport

and have often added security with DTLS [RFC6347]. Extending QUIC to support transmitting unreliable application data provides another option for secure datagrams with the added benefit of sharing the cryptographic and authentication context used for reliable streams.

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

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 want to use both a reliable stream and an unreliable flow to the same peer can benefit by sharing a single handshake and authentication context between a reliable QUIC stream and a flow of unreliable QUIC datagrams. This can reduce the latency required for handshakes compared to opening both a TLS connection and a DTLS connection.
- * QUIC uses a more nuanced loss recovery mechanism than the DTLS handshake. This can allow loss recovery to occur more quickly for QUIC data.
- * QUIC datagrams are subject to QUIC congestion control. Providing a single congestion control for both reliable and unreliable data can be more effective and efficient.

These features 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 in addition to reliable streams.

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=0x20). 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.

The default for this parameter is 0, which indicates that the endpoint does not support DATAGRAM frames. A value greater than 0 indicates that the endpoint supports the DATAGRAM frame types and is willing to receive such frames on this connection.

An endpoint MUST NOT send DATAGRAM frames until it has received the max_datagram_frame_size transport parameter with a non-zero value during the handshake (or during a previous handshake if 0-RTT is used). An endpoint MUST NOT send DATAGRAM frames that are larger than the max_datagram_frame_size value it has received from its peer. An endpoint that receives a DATAGRAM frame when it has not indicated support via the transport parameter MUST terminate the connection with an error of type `PROTOCOL_VIOLATION`. Similarly, an endpoint that receives a DATAGRAM frame that is larger than the value it sent in its max_datagram_frame_size transport parameter MUST terminate the connection with an error of type `PROTOCOL_VIOLATION`.

For most uses of DATAGRAM frames, it is RECOMMENDED to send a value of 65535 in the max_datagram_frame_size transport parameter to indicate 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 than 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 than 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 absence of the `max_datagram_frame_size` transport parameter. 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 Types

DATAGRAM frames are used to transmit application data in an unreliable manner. The Type field in the DATAGRAM frame takes the form `0b0011000X` (or the values `0x30` and `0x31`). The least significant bit of the Type field in the DATAGRAM frame is the LEN bit (`0x01`), which indicates whether 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.

DATAGRAM frames are structured as follows:

```
DATAGRAM Frame {  
  Type (i) = 0x30..0x31,  
  [Length (i)],  
  Datagram Data (...),  
}
```

Figure 1: DATAGRAM Frame Format

DATAGRAM frames contain the following fields:

Length: A variable-length integer specifying the length of the Datagram Data field in bytes. This field is present only when the LEN bit is set to 1. When the LEN bit is set to 0, the Datagram Data field 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 a 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 (as determined by factors like congestion control, described below) 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.

Like STREAM frames, DATAGRAM frames contain application data and MUST be protected with either 0-RTT or 1-RTT keys.

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_udp_payload_size` transport parameter and 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. Multiplexing Datagrams

DATAGRAM frames belong to a QUIC connection as a whole, and are not associated with any stream ID at the QUIC layer. However, it is expected that applications will want to differentiate between specific DATAGRAM frames by using identifiers, such as for logical flows of datagrams or to distinguish between different kinds of datagrams.

Identifiers used to multiplex different kinds of datagrams, or flows of datagrams, are the responsibility of the application protocol running over QUIC to define. The application defines the semantics of the Datagram Data field and how it is parsed.

If the application needs to support the coexistence of multiple flows of datagrams, one recommended pattern is to use a variable-length integer at the beginning of the Datagram Data field. This is a simple approach that allows a large number of flows to be encoded using minimal space.

QUIC implementations SHOULD present an API to applications to assign relative priorities to DATAGRAM frames with respect to each other and to QUIC streams.

5.2. Acknowledgement Handling

Although DATAGRAM frames are not retransmitted upon loss detection, they are ack-eliciting ([RFC9002]). Receivers SHOULD support delaying ACK frames (within the limits specified by `max_ack_delay`) in response to receiving packets that only contain DATAGRAM frames, since the sender takes no action if these packets are temporarily unacknowledged. Receivers will continue to send ACK frames when conditions indicate a packet might be lost, since the packet's payload is unknown to the receiver, and when dictated by `max_ack_delay` or other protocol components.

As with any ack-eliciting frame, when a sender suspects that a packet containing only DATAGRAM frames has been lost, it sends probe packets to elicit a faster acknowledgement as described in Section 6.2.4 of [RFC9002].

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 sender application that the datagram was successfully transmitted and received. Due to reordering, this can include a DATAGRAM frame that was thought to be lost, but which at a later point was received and acknowledged. It is important to note that acknowledgement of a DATAGRAM frame only indicates that the transport-layer handling on the receiver processed the frame, and does not guarantee that the application on the receiver successfully processed the data. Thus, this signal cannot replace application-layer signals that indicate successful processing.

5.3. 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 might not be able to commit the necessary resources to process the frames. For example, it might 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.

5.4. Congestion Control

DATAGRAM frames employ the QUIC connection's congestion controller. As a result, a connection might be unable to send a DATAGRAM frame generated by the application until the congestion controller allows it [RFC9002]. The sender **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 that use packet pacing (Section 7.7 of [RFC9002]) can also delay the sending of DATAGRAM frames to maintain consistent packet pacing.

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, and the security considerations of [RFC9000] apply accordingly. All application data transmitted with the DATAGRAM frame, like the STREAM frame, **MUST** be protected either by 0-RTT or 1-RTT keys.

Application protocols that allow DATAGRAM frames to be sent in 0-RTT require a profile that defines acceptable use of 0-RTT; see Section 5.6 of [RFC9001].

The use of DATAGRAM frames might be detectable by an adversary on path that is capable of dropping packets. Since DATAGRAM frames do not use transport-level retransmission, connections that use DATAGRAM frames might be distinguished from other connections due to their different response to packet loss.

7. IANA Considerations

7.1. QUIC Transport Parameter

This document registers a new value in the QUIC Transport Parameter Registry maintained at <https://www.iana.org/assignments/quic/quic.xhtml#quic-transport>.

Value: 0x20

Parameter Name: max_datagram_frame_size

Status: permanent

Specification: This document

7.2. QUIC Frame Types

This document registers two new values in the QUIC Frame Type registry maintained at <https://www.iana.org/assignments/quic/quic.xhtml#quic-frame-types>.

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

Frame Name: DATAGRAM

Status: permanent

Specification: This document

8. Acknowledgments

The original proposal for this work came from Ian Swett.

This document had reviews and input from many contributors in the IETF QUIC Working Group, with substantive input from Nick Banks, Lucas Pardue, Rui Paulo, Martin Thomson, Victor Vasiliev, and Chris Wood.

9. References

9.1. Normative References

- [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/rfc/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/rfc/rfc8174>>.
- [RFC9000] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <<https://www.rfc-editor.org/rfc/rfc9000>>.
- [RFC9001] Thomson, M., Ed. and S. Turner, Ed., "Using TLS to Secure QUIC", RFC 9001, DOI 10.17487/RFC9001, May 2021, <<https://www.rfc-editor.org/rfc/rfc9001>>.
- [RFC9002] Iyengar, J., Ed. and I. Swett, Ed., "QUIC Loss Detection and Congestion Control", RFC 9002, DOI 10.17487/RFC9002, May 2021, <<https://www.rfc-editor.org/rfc/rfc9002>>.

9.2. Informative References

- [RFC0768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, DOI 10.17487/RFC0768, August 1980, <<https://www.rfc-editor.org/rfc/rfc768>>.
- [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/rfc/rfc6347>>.

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: 29 September 2022

M. Duke
Google
N. Banks
Microsoft
C. Huitema
Private Octopus Inc.
28 March 2022

QUIC-LB: Generating Routable QUIC Connection IDs
draft-ietf-quic-load-balancers-13

Abstract

QUIC address migration allows clients to change their IP address while maintaining connection state. To reduce the ability of an observer to link two IP addresses, clients and servers use new connection IDs when they communicate via different client addresses. This poses a problem for traditional "layer-4" load balancers that route packets via the IP address and port 4-tuple. This specification provides a standardized means of securely encoding routing information in the server's connection IDs so that a properly configured load balancer can route packets with migrated addresses correctly. As it proposes a structured connection ID format, it also provides a means of connection IDs self-encoding their length to aid some hardware offloads.

Note to Readers

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/quicwg/load-balancers> (<https://github.com/quicwg/load-balancers>).

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 29 September 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction	4
1.1. Terminology	5
1.2. Notation	5
2. First CID octet	6
2.1. Config Rotation	6
2.2. Configuration Failover	7
2.3. Length Self-Description	7
2.4. Format	7
3. Load Balancing Preliminaries	8
3.1. Unroutable Connection IDs	8
3.2. Fallback Algorithms	10
3.3. Server ID Allocation	10
4. Server ID Encoding in Connection IDs	11
4.1. CID format	11
4.2. Configuration Agent Actions	11
4.3. Server Actions	11
4.3.1. Special Case: Single Pass Encryption	12
4.3.2. General Case: Four-Pass Encryption	12
4.4. Load Balancer Actions	14
4.4.1. Special Case: Single Pass Encryption	15
4.4.2. General Case: Four-Pass Encryption	15
5. Per-connection state	15
6. Additional Use Cases	16
6.1. Load balancer chains	16
6.2. Moving connections between servers	17

7. Version Invariance of QUIC-LB	17
8. Security Considerations	18
8.1. Attackers not between the load balancer and server	19
8.2. Attackers between the load balancer and server	19
8.3. Multiple Configuration IDs	19
8.4. Limited configuration scope	19
8.5. Stateless Reset Oracle	20
8.6. Connection ID Entropy	21
9. IANA Considerations	21
10. References	22
10.1. Normative References	22
10.2. Informative References	22
Appendix A. QUIC-LB YANG Model	23
A.1. Tree Diagram	29
Appendix B. Load Balancer Test Vectors	29
B.1. Unencrypted CIDs	30
B.2. Encrypted CIDs	30
Appendix C. Interoperability with DTLS over UDP	30
C.1. DTLS 1.0 and 1.2	30
C.2. DTLS 1.3	31
C.3. Future Versions of DTLS	32
Appendix D. Acknowledgments	32
Appendix E. Change Log	32
E.1. since draft-ietf-quic-load-balancers-12	32
E.2. since draft-ietf-quic-load-balancers-11	32
E.3. since draft-ietf-quic-load-balancers-10	32
E.4. since draft-ietf-quic-load-balancers-09	33
E.5. since draft-ietf-quic-load-balancers-08	33
E.6. since draft-ietf-quic-load-balancers-07	33
E.7. since draft-ietf-quic-load-balancers-06	33
E.8. since draft-ietf-quic-load-balancers-05	33
E.9. since draft-ietf-quic-load-balancers-04	34
E.10. since draft-ietf-quic-load-balancers-03	34
E.11. since draft-ietf-quic-load-balancers-02	34
E.12. since draft-ietf-quic-load-balancers-01	34
E.13. since draft-ietf-quic-load-balancers-00	35
E.14. Since draft-duke-quic-load-balancers-06	35
E.15. Since draft-duke-quic-load-balancers-05	35
E.16. Since draft-duke-quic-load-balancers-04	35
E.17. Since draft-duke-quic-load-balancers-03	35
E.18. Since draft-duke-quic-load-balancers-02	35
E.19. Since draft-duke-quic-load-balancers-01	36
E.20. Since draft-duke-quic-load-balancers-00	36
Authors' Addresses	36

1. Introduction

QUIC packets [RFC9000] 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.

As this document proposes a structured QUIC Connection ID, it also proposes a system for self-encoding connection ID length in all packets, so that crypto offload can efficiently obtain 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 [RFC9000]. 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. 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."

2.1. 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 8.3).

2.2. 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. These connection IDs may be of any length; however, see Section 8.6 for limits on this length.

2.3. 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.

2.4. 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.

3. 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 2.1).

3.1. Unroutable 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. Those that are not routable are "unroutable DCIDs" and receive similar treatment regardless of why they're unroutable:

- * The config rotation bits (Section 2.1) may not correspond to an active configuration. Note: a packet with a DCID that indicates 5-tuple routing (see Section 2.2) is always routable.
- * 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 routable.

Load balancers **MUST** forward packets with routable DCIDs to a server in accordance with the chosen routing algorithm. Exception: if the load balancer can parse the QUIC packet and makes a routing decision depending on the contents (e.g., the SNI in a TLS client hello), it **MAY** route in accordance with this instead. However, load balancers **MUST** always route long header packets it cannot parse in accordance with the DCID (see Section 7).

Load balancers **SHOULD** drop short header packets with unroutable DCIDs.

When forwarding a packet with a long header and unroutable DCID, load balancers **MUST** use a fallback algorithm as specified in Section 3.2.

Load balancers **MAY** drop packets with long headers and unroutable DCIDs if and only if it knows that the encoded QUIC version does not allow an unroutable DCID in a packet with that signature. For example, a load balancer can safely drop a QUIC version 1 Handshake packet with an unroutable DCID, as a version 1 Handshake packet sent to a QUIC-LB routable server will always have a server-generated routable 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 unroutable CIDs **MUST** use the available mechanisms to induce the client to use a routable CID in future packets. In QUIC version 1, this requires using a routable CID in the Source CID field of server-generated long headers.

3.2. Fallback Algorithms

There are conditions described below where a load balancer routes a packet using a "fallback 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 [RFC8999] to maximize its robustness to future versions of QUIC.

A fallback algorithm 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 fallback algorithm might convert a unroutable 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 7.

3.3. Server ID Allocation

Load Balancer configurations include a mapping of server IDs to forwarding addresses. The corresponding server configurations contain one or more unique server IDs.

The configuration agent chooses a server ID length for each configuration that MUST be at least one octet.

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 unroutable.

The configuration agent SHOULD provide a means for servers to express the number of server IDs it can usefully employ, because a single routing address actually corresponds to multiple server entities (see Section 6.1).

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. Server ID Encoding in Connection IDs

4.1. CID format

All connection IDs use the following format:

```
QUIC-LB Connection ID {  
    First Octet (8),  
    Server ID (8..152-len(Nonce)),  
    Nonce (32..152-len(Server ID)),  
}
```

Figure 3: CID Format

4.2. Configuration Agent Actions

The configuration agent assigns a server ID to every server in its pool in accordance with Section 3.3, and determines a server ID length (in octets) sufficiently large to encode all server IDs, including potential future servers.

Each configuration specifies the length of the Server ID and Nonce fields, with limits defined for each algorithm.

Optionally, it also defines a 16-octet key. Note that failure to define a key means that observers can determine the assigned server of any connection, significantly increasing the linkability of QUIC address migration.

The nonce length **MUST** be at least 4 octets. The server ID length **MUST** be at least 1 octet.

As QUIC version 1 limits connection IDs to 20 octets, the server ID and nonce lengths **MUST** sum to 19 octets or less.

4.3. Server Actions

The server writes the first octet and its server ID into their respective fields.

If there is no key in the configuration, the server **MUST** fill the Nonce field with bytes that appear to be random. If there is a key, the server fills the nonce field with a nonce of its choosing. See Section 8.6 for details.

The server **MAY** append additional bytes to the connection ID, up to the limit specified in that version of QUIC, for its own use. These bytes **MUST NOT** provide observers with any information that could link

two connection IDs to the same connection, client, or server. In particular, all servers using a configuration MUST consistently add the same length to each connection ID, to preserve the linkability objectives of QUIC-LB. Any additional bytes SHOULD appear random unless individual servers are not distinguishable (e.g. any server using that configuration appends identical bytes to every connection ID).

If there is no key in the configuration, the Connection ID is complete. Otherwise, there are further steps, as described in the two following subsections.

Encryption 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.

4.3.1. Special Case: Single Pass Encryption

When the nonce length and server ID length sum to exactly 16 octets, the server MUST use a single-pass encryption algorithm. All connection ID octets except the first form an AES-ECB block. This block is encrypted once, and the result forms the second through seventeenth most significant bytes of the connection ID.

4.3.2. General Case: Four-Pass Encryption

Any other field length requires four passes for encryption and at least three for decryption. To understand this algorithm, it is useful to define four functions that minimize the amount of bit-shifting necessary in the event that there are an odd number of octets.

The `expand_left()` function outputs 16 octets, with its first argument in the most significant bits, its second argument in the least significant byte, and zeros in all other positions. Thus,

```
expand_left(0xaaba3c, 0x13) = 0xaaba3c00000000000000000000000013
```

`expand_right()` is similar, except that the second argument is in the most significant byte, and the first argument is in the least significant bits. Therefore,

```
expand_right(0xaaba3c, 0x13) = 0x13000000000000000000000000aaba3c
```

Similarly, `truncate_left()` and `truncate_right()` take the most significant and least significant bits, respectively, from a ciphertext. For example, to take 28 bits of a ciphertext:

```
truncate_left(0x2094842ca49256198c2deaa0ba53caa0, 28) = 0x2094842
truncate_right(0x2094842ca49256198c2deaa0ba53caa0, 28) = 0xa53caa0
```

The example at the end of this section helps to clarify the steps described below.

1. The server concatenates the server ID and nonce to create plaintext_CID.
2. The server splits plaintext_CID into components left_0 and right_0 of equal length, splitting an odd octet in half if necessary. For example, 0x7040b81b55ccf3 would split into a left_0 of 0x7040b81 and right_0 of 0xb55ccf3.
3. Encrypt the result of expand_left(left_0) to obtain a ciphertext.
4. XOR the least significant bits of the ciphertext with right_0 to form right_1.

```
Thus steps 3 and 4 can be expressed as right_1 = right_0 ^
truncate_right( AES_ECB(key, expand_left(left_0, 0x01)),
len(right_0))
```

5. Repeat steps 3 and 4, but use them to compute left_1 by expanding and encrypting right_1 with the most significant octet as 0x02 and XOR the results with left_0.

```
left_1 = left_0 ^ truncate_left( AES_ECB(key,
expand_right(right_1, 0x02)), len(left_0))
```

6. Repeat steps 3 and 4, but use them to compute right_2 by expanding and encrypting left_1 with the least significant octet as 0x03 and XOR the results with right_1.

```
right_2 = right_1 ^ truncate_right( AES_ECB(key,
expand_left(left_1, 0x03)), len(right_1))
```

7. Repeat steps 3 and 4, but use them to compute left_2 by expanding and encrypting right_2 with the most significant octet as 0x04 and XOR the results with left_1.

```
left_2 = left_1 ^ truncate_left( AES_ECB(key,
expand_right(right_2, 0x04)), len(left_1))
```

8. The server concatenates left_2 with right_2 to form the ciphertext CID, which it appends to the first octet.

The following example executes the steps for the provided inputs. Note that the plaintext is of odd octet length, so the middle octet will be split evenly left_0 and right_0.

```
server_id = 0x31441a
nonce = 0x9c69c275
key = 0xdf726a9893ec05c0632d3956680baf0

// step 1
plaintext_CID = 0x31441a9c69c275

// step 2
left_0 = 0x31441a9
right_0 = 0xc69c275

// step 3
aes_input = 0x31441a90000000000000000000000001
ciphertext = 0x4d140de42d0b85bdf554ba35c1d5c653

// step 4
right_1 = 0xc69c275 ^ 0x1d5c653 = 0xdbbc0426

// step 5
aes_input = 0x020000000000000000000000dbbc0426
aes_output = 0x7e99160f3cf5b89c70584ccd2c2cd24b
left_1 = 0x31441a9 ^ 0x7e99160 = 0x4fdd0c9

// step 6
AES input = 0x4fdd0c90000000000000000000000003
AES output = 0x26c1d5a3a5e31ff8e3ca505da6061ac6
right_2 = 0xdbbc0426 ^ 0x6061ac6 = 0xbba1ee0

// step 7
AES input = 0x04000000000000000000000000bba1ee0
AES output = 0xad1b8b25b436a94007d80cf3704377b
left_2 = 0x4fdd0c9 ^ 0xad1b8b = 0xe23cb42

// step 8
cid = first_octet || left_2 || right_2 = 0x07e23cb42bba1ee0
```

4.4. Load Balancer Actions

On each incoming packet, the load balancer extracts consecutive octets, beginning with the second octet. If there is no key, the first octets correspond to the server ID.

If there is a key, the load balancer takes one of two actions:

4.4.1. Special Case: Single Pass Encryption

If server ID length and nonce length sum to exactly 16 octets, they form a ciphertext block. The load balancer decrypts the block using the AES-ECB key and extracts the server ID from the most significant bytes of the resulting plaintext.

4.4.2. General Case: Four-Pass Encryption

First, split the ciphertext CID (excluding the first octet) into its equal-length components `left_2` and `right_2`. Then follow the process below:

```
left_1 = left_2 ^ truncate_left(AES_ECB(key, expand_right(right_2), 0x04))
right_1 = right_2 ^ truncate_right(AES_ECB(key, expand_left(left_1, 0x03))
left_0 = left_1 ^ truncate_left(AES_ECB(key, expand_right(right_1), 0x02))
```

As the load balancer has no need for the nonce, it can conclude after 3 passes as long as the server ID is entirely contained in `left_0` (i.e., the nonce is at least as large as the server ID). If the server ID is longer, a fourth pass is necessary:

```
right_0 = right_1 ^ truncate_right(AES_ECB(key, expand_left(left_0,
0x01)))
```

and the load balancer has to concatenate `left_0` and `right_0` to obtain the complete server ID.

5. Per-connection state

QUIC-LB requires no per-connection state at the load balancer. The load balancer can extract the server ID from the connection ID of each incoming packet and route that packet accordingly.

However, once the routing decision has been made, the load balancer MAY associate the 4-tuple with the decision. This has two advantages:

- * The load balancer only extracts the server ID once per incoming 4-tuple. When the CID is encrypted, this substantially reduces computational load.
- * Incoming Stateless Reset packets and ICMP messages are easily routed to the correct origin server.

In addition to the increased state requirements, however, load balancers cannot detect the CONNECTION_CLOSE frame to indicate the end of the connection, so they rely on a timeout to delete connection state. There are numerous considerations around setting such a timeout.

In the event a connection ends, freeing an IP and port, and a different connection migrates to that IP and port before the timeout, the load balancer will misroute the different connection's packets to the original server. A short timeout limits the likelihood of such a misrouting.

Furthermore, if a short timeout causes premature deletion of state, the routing is easily recoverable by decoding an incoming Connection ID. However, a short timeout also reduces the chance that an incoming Stateless Reset is correctly routed.

Servers MAY implement the technique described in Section 14.4.1 of [RFC9000] in case the load balancer is stateless, 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.

6. Additional Use Cases

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

6.1. 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.

If the number of necessary server IDs per next hop is uniform, a simple implementation would use successively longer server IDs at each tier of load balancing, and the server configuration would match the last tier. The forward load balancers would simply treat the least significant bits of the server ID as part of the nonce.

6.2. 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.

7. Version Invariance of QUIC-LB

The server ID encodings, and requirements for their handling, are designed to be QUIC version independent (see [RFC8999]). 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 3.1 provides guidance about how load balancers should handle unroutable 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.

While this document does not update the commitments in [RFC8999], 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 unroutable 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. See also Section 3.1 for other considerations about this case. Note that an SNI-aware load balancer, faced with an unknown QUIC version, might misdirect initial packets to the wrong tenant. While inefficient, this preserves the ability for tenants to deploy new versions provided they have an out-of-band means of providing a connection ID for the client to use.

8. Security Considerations

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

Note that without a key for the encoding, QUIC-LB makes no attempt to obscure the server mapping, and therefore does not address these concerns. Without a key, QUIC-LB merely allows consistent CID encoding for compatibility across a network infrastructure, which makes QUIC robust to NAT rebinding. Servers that are encoding their server ID without a key 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 2.1. 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.

8.1. 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.

An encrypted encoding provides robust protection against this. An unencrypted one provides none.

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

8.2. 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.

8.3. Multiple Configuration IDs

During the period in which there are multiple deployed configuration IDs (see Section 2.1), 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.

8.4. 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. In many cases, the load balancers can distinguish these configurations by external IP address.

However, assigning multiple entities to an IP address is complimentary with concealing DNS requests (e.g., DoH [RFC8484]) and the TLS Server Name Indicator (SNI) ([I-D.ietf-tls-esni]) to obscure the ultimate destination of traffic. While the load balancer's

fallback algorithm (Section 3.2) can use the SNI to make a routing decision on the first packet, there are three ways to route subsequent packets:

- * all co-tenants can use the same QUIC-LB configuration, leaking the server mapping to each other as described above;
- * co-tenants can be issued one of up to three configurations distinguished by the config rotation bits (Section 2.1), exposing information about the target domain to the entire network; or
- * tenants can use 4-tuple routing in their CIDs (in which case they SHOULD disable migration in their connections), which neutralizes the value of QUIC-LB but preserves privacy.

When configuring QUIC-LB, administrators must evaluate the privacy tradeoff considering the relative value of each of these properties, given the trust model between tenants, the presence of methods to obscure the domain name, and value of address migration in the tenant use cases.

As the plaintext algorithm makes no attempt to conceal the server mapping, these deployments SHOULD simply use a common configuration.

8.5. Stateless Reset Oracle

Section 21.9 of [RFC9000] 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.

Note also that when a server starts using a new QUIC-LB config rotation codepoint, new CIDs might not be unique with respect to previous configurations that occupied that codepoint, and therefore different clients may have observed the same CID and stateless reset token. A straightforward method of managing stateless reset keys is to maintain a separate key for each config rotation codepoint, and replace each key when the configuration for that codepoint changes. Thus, a server transitions from one config to another, it will be able to generate correct tokens for connections using either type of CID.

8.6. Connection ID Entropy

If a server ever reuses a nonce in generating a CID for a given configuration, it risks exposing sensitive information. Given the same server ID, the CID will be identical (aside from a possible difference in the first octet). This can risk exposure of the QUIC-LB key. If two clients receive the same connection ID, they also have each other's stateless reset token unless that key has changed in the interim.

The encrypt mode needs 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 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 maximize entropy, servers SHOULD start with a random nonce value, in which case the configuration is usable until the nonce value wraps around to zero and then reaches the initial value again.

Whether or not it implements the counter method, the server MUST NOT reuse a nonce until it switches to a configuration with new keys.

If the nonce is sent in plaintext, servers MUST generate nonces so that they appear to be random. Observable correlations between plaintext nonces would provide trivial linkability between individual connections, rather than just to a common server.

For any algorithm, 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.

When sizing a nonce that is to be randomly generated, the configuration agent SHOULD consider that a server generating a N-bit nonce will create a duplicate about every $2^{(N/2)}$ attempts, and therefore compare the expected rate at which servers will generate CIDs with the lifetime of a configuration.

9. IANA Considerations

There are no IANA requirements.

10. References

10.1. Normative References

- [RFC8999] Thomson, M., "Version-Independent Properties of QUIC", RFC 8999, DOI 10.17487/RFC8999, May 2021, <<https://www.rfc-editor.org/info/rfc8999>>.
- [RFC9000] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <<https://www.rfc-editor.org/info/rfc9000>>.

10.2. Informative References

- [I-D.draft-ietf-tls-dtls13]
Rescorla, E., Tschofenig, H., and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3", Work in Progress, Internet-Draft, draft-ietf-tls-dtls13-43, 30 April 2021, <<https://www.ietf.org/archive/id/draft-ietf-tls-dtls13-43.txt>>.
- [I-D.ietf-tls-dtls-connection-id]
Rescorla, E., Tschofenig, H., Fossati, T., and A. Kraus, "Connection Identifier for DTLS 1.2", Work in Progress, Internet-Draft, draft-ietf-tls-dtls-connection-id-13, 22 June 2021, <<https://www.ietf.org/archive/id/draft-ietf-tls-dtls-connection-id-13.txt>>.
- [I-D.ietf-tls-esni]
Rescorla, E., Oku, K., Sullivan, N., and C. A. Wood, "TLS Encrypted Client Hello", Work in Progress, Internet-Draft, draft-ietf-tls-esni-14, 13 February 2022, <<https://www.ietf.org/archive/id/draft-ietf-tls-esni-14.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>>.
- [RFC4347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security", RFC 4347, DOI 10.17487/RFC4347, April 2006, <<https://www.rfc-editor.org/info/rfc4347>>.

- [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>>.
- [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>>.
- [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>>.
- [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/info/rfc7983>>.
- [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>>.
- [RFC8484] Hoffman, P. and P. McManus, "DNS Queries over HTTPS (DoH)", RFC 8484, DOI 10.17487/RFC8484, October 2018, <<https://www.rfc-editor.org/info/rfc8484>>.

Appendix A. QUIC-LB YANG Model

These YANG models conform to [RFC6020] and express a complete QUIC-LB configuration. There is one model for the server and one for the middlebox (i.e the load balancer and/or Retry Service).

```
module ietf-quic-lb-server {
  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)
           Christian Huitema (huitema at huitema.net)";

description
  "This module enables the explicit cooperation of QUIC servers with
  trusted intermediaries without breaking important protocol
  features.

  Copyright (c) 2022 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 "2022-02-11" {
  description
    "Updated to design in version 13 of the draft";
  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";
}

leaf config-id {
  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 server-id-length {
  type uint8 {
    range "1..15";
  }
  must '. <= (19 - ../nonce-length)' {
    error-message
      "Server ID and nonce lengths must sum to no more than 19.";
  }
  mandatory true;
  description
    "Length (in octets) of a server ID. Further range-limited
    by nonce-length.";
}

leaf nonce-length {
  type uint8 {
    range "4..18";
  }
  mandatory true;
  description
```

```
        "Length, in octets, of the nonce. Short nonces mean there will
        be frequent configuration updates.";
    }

    leaf cid-key {
        type quic-lb-key;
        description
            "Key for encrypting the connection ID.";
    }

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

module ietf-quic-lb-middlebox {
    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)
                 Christian Huitema (huitema at huitema.net)";
```

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 "2021-02-11" {  
  description  
    "Updated to design in version 13 of the draft";  
  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 server-id-length {
  type uint8 {
    range "1..15";
  }
  must '. <= (19 - ../nonce-length)' {
    error-message
      "Server ID and nonce lengths must sum to no more than 19.";
  }
  mandatory true;
  description
    "Length (in octets) of a server ID. Further range-limited
    by nonce-length.";
}

leaf cid-key {
  type quic-lb-key;
  description
    "Key for encrypting the connection ID.";
}

leaf nonce-length {
  type uint8 {
    range "4..18";
  }
  mandatory true;
  description
    "Length, in octets, of the nonce. Short nonces mean there
    will be frequent configuration updates.";
}

list server-id-mappings {
  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";
    }
}
}
}
}

```

A.1. Tree Diagram

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

```

module: ietf-quic-lb-server
  +--rw quic-lb!
    +--rw config-id                               uint8
    +--rw first-octet-encodes-cid-length?         boolean
    +--rw server-id-length                       uint8
    +--rw nonce-length                           uint8
    +--rw cid-key?                               quic-lb-key
    +--rw server-id                             yang:hex-string

module: ietf-quic-lb-middlebox
  +--rw quic-lb!
    +--rw cid-configs* [config-rotation-bits]
      +--rw config-rotation-bits                 uint8
      +--rw server-id-length                     uint8
      +--rw cid-key?                             quic-lb-key
      +--rw nonce-length                         uint8
      +--rw server-id-mappings* [server-id]
        +--rw server-id                         yang:hex-string
        +--rw server-address                     inet:ip-address

```

Appendix B. Load Balancer Test Vectors

This section uses the following abbreviations:

cid Connection ID
cr_bits Config Rotation Bits
LB Load Balancer
sid Server ID

In all cases, the server is configured to encode the CID length.

B.1. Unencrypted CIDs

```
cr_bits sid nonce cid
0 c4605e 4504cc4f 07c4605e4504cc4f
1 350d28b420 3487d970b 40a350d28b4203487d970b
```

B.2. Encrypted CIDs

The key for all of these examples is 8f95f09245765f80256934e50c66207f. The test vectors include an example that uses the 16-octet single-pass special case, as well as an instance where the server ID length exceeds the nonce length, requiring a fourth decryption pass.

```
cr_bits sid nonce cid
0 ed793a ee080dbf 07fbfe05f731b425
1 ed793a51d49b8f5fab65 ee080dbf48 4f010956fb5c1d4d86e010183e0b7dle
2 ed793a51d49b8f5f ee080dbf48c0dle5 904dd2d05a7b0de9b2b9907afb5ecf8cc3
0 ed793a51d49b8f5fab ee080dbf48c0dle55d 127a285a09f85280f4fd6abb434a7159e4d3eb
```

Appendix C. Interoperability with DTLS over UDP

Some environments may contain DTLS traffic as well as QUIC operating over UDP, which may be hard to distinguish.

In most cases, the packet parsing rules above will cause a QUIC-LB load balancer to route DTLS traffic in an appropriate way. DTLS 1.3 implementations that use the connection_id extension [I-D.ietf-tls-dtls-connection-id] might use the techniques in this document to generate connection IDs and achieve robust routability for DTLS associations if they meet a few additional requirements. This non-normative appendix describes this interaction.

C.1. DTLS 1.0 and 1.2

DTLS 1.0 [RFC4347] and 1.2 [RFC6347] use packet formats that a QUIC-LB router will interpret as short header packets with CIDs that request 4-tuple routing. As such, they will route such packets consistently as long as the 4-tuple does not change. Note that DTLS 1.0 has been deprecated by the IETF.

The first octet of every DTLS 1.0 or 1.2 datagram contains the content type. A QUIC-LB load balancer will interpret any content type less than 128 as a short header packet, meaning that the subsequent octets should contain a connection ID.

Existing TLS content types comfortably fit in the range below 128. Assignment of codepoints greater than 64 would require coordination in accordance with [RFC7983], and anyway would likely create problems demultiplexing DTLS and version 1 of QUIC. Therefore, this document believes it is extremely unlikely that TLS content types of 128 or greater will be assigned. Nevertheless, such an assignment would cause a QUIC-LB load balancer to interpret the packet as a QUIC long header with an essentially random connection ID, which is likely to be routed irregularly.

The second octet of every DTLS 1.0 or 1.2 datagram is the bitwise complement of the DTLS Major version (i.e. version 1.x = 0xfe). A QUIC-LB load balancer will interpret this as a connection ID that requires 4-tuple based load balancing, meaning that the routing will be consistent as long as the 4-tuple remains the same.

[I-D.ietf-tls-dtls-connection-id] defines an extension to add connection IDs to DTLS 1.2. Unfortunately, a QUIC-LB load balancer will not correctly parse the connection ID and will continue 4-tuple routing. A modified QUIC-LB load balancer that correctly identifies DTLS and parses a DTLS 1.2 datagram for the connection ID is outside the scope of this document.

C.2. DTLS 1.3

DTLS 1.3 [I-D.draft-ietf-tls-dtls13] changes the structure of datagram headers in relevant ways.

Handshake packets continue to have a TLS content type in the first octet and 0xfe in the second octet, so they will be 4-tuple routed, which should not present problems for likely NAT rebinding or address change events.

Non-handshake packets always have zero in their most significant bit and will therefore always be treated as QUIC short headers. If the connection ID is present, it follows in the succeeding octets. Therefore, a DTLS 1.3 association where the server utilizes Connection IDs and the encodings in this document will be routed correctly in the presence of client address and port changes.

However, if the client does not include the `connection_id` extension in its ClientHello, the server is unable to use connection IDs. In this case, non-handshake packets will appear to contain random

connection IDs and be routed randomly. Thus, unmodified QUIC-LB load balancers will not work with DTLS 1.3 if the client does not advertise support for connection IDs, or the server does not request the use of a compliant connection ID.

A QUIC-LB load balancer might be modified to identify DTLS 1.3 packets and correctly parse the fields to identify when there is no connection ID and revert to 4-tuple routing, removing the server requirement above. However, such a modification is outside the scope of this document, and classifying some packets as DTLS might be incompatible with future versions of QUIC.

C.3. Future Versions of DTLS

As DTLS does not have an IETF consensus document that defines what parts of DTLS will be invariant in future versions, it is difficult to speculate about the applicability of this section to future versions of DTLS.

Appendix D. Acknowledgments

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

Appendix E. Change Log

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

E.1. since draft-ietf-quic-load-balancers-12

- * Separated Retry Service design into a separate draft.

E.2. since draft-ietf-quic-load-balancers-11

- * Fixed mistakes in test vectors

E.3. since draft-ietf-quic-load-balancers-10

- * Refactored algorithm descriptions; made the 4-pass algorithm easier to implement
- * Revised test vectors
- * Split YANG model into a server and middlebox version

E.4. since draft-ietf-quic-load-balancers-09

- * Renamed "Stream Cipher" and "Block Cipher" to "Encrypted Short" and "Encrypted Long"
- * Added section on per-connection state
- * Changed "Encrypted Short" to a 4-pass algorithm.
- * Recommended a random initial nonce when incrementing.
- * Clarified what SNI LBs should do with unknown QUIC versions.

E.5. since draft-ietf-quic-load-balancers-08

- * Eliminate Dynamic SID allocation
- * Eliminated server use bytes

E.6. since draft-ietf-quic-load-balancers-07

- * Shortened SSCID nonce minimum length to 4 bytes
- * Removed RSCID from Retry token body
- * Simplified CID formats
- * Shrunk size of SID table

E.7. since draft-ietf-quic-load-balancers-06

- * Added interoperability with DTLS
- * Changed "non-compliant" to "unroutable"
- * Changed "arbitrary" algorithm to "fallback"
- * Revised security considerations for mistrustful tenants
- * Added retry service considerations for non-Initial packets

E.8. 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
- E.9. 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
- E.10. since-draft-ietf-quic-load-balancers-03
- * Improved Config Rotation text
 - * Added stream cipher test vectors
 - * Deleted the Obfuscated CID algorithm
- E.11. 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
- E.12. 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
- E.13. since-draft-ietf-quic-load-balancers-00
- * Removed in-band protocol from the document
- E.14. Since draft-duke-quic-load-balancers-06
- * Switch to IETF WG draft.
- E.15. 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
- E.16. Since draft-duke-quic-load-balancers-04
- * Added standard for retry services
- E.17. 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
- E.18. 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

E.19. Since draft-duke-quic-load-balancers-01

- * Complete rewrite
- * Supports multiple security levels
- * Lightweight messages

E.20. Since draft-duke-quic-load-balancers-00

- * Converted to markdown
- * Added variable length connection IDs

Authors' Addresses

Martin Duke
Google
Email: martin.h.duke@gmail.com

Nick Banks
Microsoft
Email: nibanks@microsoft.com

Christian Huitema
Private Octopus Inc.
Email: huitema@huitema.net

Network Working Group
Internet-Draft
Intended status: Informational
Expires: 8 October 2022

M. Kuehlewind
Ericsson
B. Trammell
Google Switzerland GmbH
6 April 2022

Manageability of the QUIC Transport Protocol
draft-ietf-quic-manageability-16

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. It is intended as a "user's manual" for the wire image, providing guidance for 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 8 October 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction	3
2. Features of the QUIC Wire Image	4
2.1. QUIC Packet Header Structure	4
2.2. Coalesced Packets	6
2.3. Use of Port Numbers	7
2.4. The QUIC Handshake	7
2.5. Integrity Protection of the Wire Image	12
2.6. Connection ID and Rebinding	12
2.7. Packet Numbers	13
2.8. Version Negotiation and Greasing	13
3. Network-Visible Information about QUIC Flows	14
3.1. Identifying QUIC Traffic	14
3.1.1. Identifying Negotiated Version	15
3.1.2. First Packet Identification for Garbage Rejection	15
3.2. Connection Confirmation	15
3.3. Distinguishing Acknowledgment Traffic	16
3.4. Server Name Indication (SNI)	16
3.4.1. Extracting Server Name Indication (SNI) Information	16
3.5. Flow Association	18
3.6. Flow Teardown	18
3.7. Flow Symmetry Measurement	19
3.8. Round-Trip Time (RTT) Measurement	19
3.8.1. Measuring Initial RTT	19
3.8.2. Using the Spin Bit for Passive RTT Measurement	19
4. Specific Network Management Tasks	21
4.1. Passive Network Performance Measurement and Troubleshooting	21
4.2. Stateful Treatment of QUIC Traffic	22
4.3. Address Rewriting to Ensure Routing Stability	23
4.4. Server Cooperation with Load Balancers	24
4.5. Filtering Behavior	24
4.6. UDP Blocking, Throttling, and NAT Binding	24
4.7. DDoS Detection and Mitigation	25
4.8. Quality of Service Handling and ECMP Routing	27
4.9. Handling ICMP Messages	27
4.10. Guiding Path MTU	27

5. IANA Considerations	29
6. Security Considerations	29
7. Contributors	29
8. Acknowledgments	30
9. References	30
9.1. Normative References	30
9.2. Informative References	31
Authors' Addresses	34

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 mutable by the network. This is enforced through integrity protection of the wire image [WIRE-IMAGE]. Encryption of most transport-layer control signaling means that less information is visible to the network than is the case with TCP.

Integrity protection can also simplify troubleshooting at the end points as none of the nodes on the network path can modify transport layer information. However, it means in-network operations that depend on modification of data (for examples, see [RFC9065]) are not possible without the cooperation of a QUIC endpoint. Such cooperation 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. The presence of a particular practice in this document should therefore not be construed as a recommendation to apply it. For each practice, this document describes what is and is not possible with the QUIC transport protocol as defined.

This document focuses solely on network management practices that observe traffic on the wire. Replacement of troubleshooting based on observation with active measurement techniques, for example, is therefore out of scope. A more generalized treatment of network management operations on encrypted transports is given in [RFC9065].

QUIC-specific terminology used in this document is defined in [QUIC-TRANSPORT].

2. Features of the QUIC Wire Image

This section discusses those aspects of the QUIC transport protocol that have an impact on the design and operation of devices that forward QUIC packets. This section is therefore primarily considering the unencrypted part of QUIC's wire image [WIRE-IMAGE], which is defined 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 addresses 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.

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. It contains a version number, as well as source and destination connection IDs for associating packets with a QUIC connection. 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].

In version 1 of QUIC, the long header is used during connection establishment to transmit crypto handshake data, perform version negotiation, retry, and send 0-RTT data.

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 ([RFC7801]). All deployed versions are maintained in an IANA registry (see Section 22.2 of [QUIC-TRANSPORT]).
- * **source and destination connection ID:** short and long 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 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 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 protected 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 can be decoded in long headers, while it is implicit (depending on destination connection ID length) in short headers. The length of the packet number is cryptographically protected.
- * 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 protected.

2.2. Coalesced Packets

Multiple QUIC packets may be coalesced into a single 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 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 the HTTP Alternative Services mechanism [RFC7838] 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 (protocol, source and destination IP address, and source and destination port). However, if the connection ID is zero-length, all packets of the 5-tuple likely 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 four flights of datagrams labelled "Client Initial", "Server Initial", "Client Completion", and "Server Completion", as illustrated in Figure 1.

A handshake starts with the client sending one or more datagrams containing Initial packets, detailed in Figure 2, which elicits the Server Initial response detailed in Figure 3 typically containing three types of packets: Initial packet(s) with the beginning of the server's side of the TLS handshake, Handshake packet(s) with the rest of the server's portion of the TLS handshake, and 1-RTT packet(s), if present.

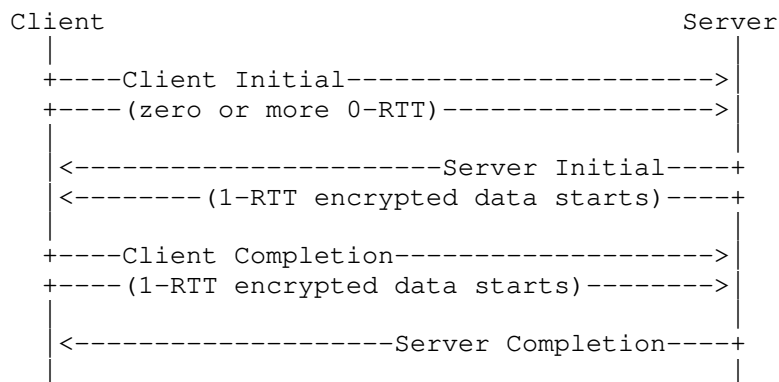


Figure 1: General communication pattern visible in the QUIC handshake

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. The Client Completion flight contains at least one Handshake packet and could also include an Initial packet. QUIC packets in separate contexts during the handshake can be coalesced (see Section 2.2) in order to reduce the number of UDP datagrams sent during the handshake.

Handshake packets can arrive out-of-order without impacting the handshake as long as the reordering was not accompanied by extensive delays that trigger a spurious Probe Timeout (Section 6.2 of RFC9002). If QUIC packets get lost or reordered, packets belonging to the same flight might not be observed in close time succession, though the sequence of the flights will not change, because one flight depends upon the peer's previous flight.

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

Initial, Handshake, and 1-RTT packets belong to different cryptographic and transport contexts. The Client Completion (Figure 4) and the Server Completion (Figure 5) flights conclude the Initial and Handshake contexts, by sending final acknowledgments and CRYPTO frames.

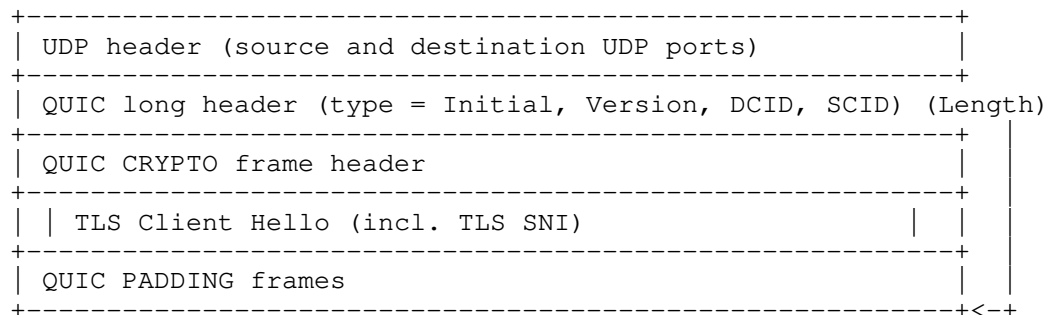


Figure 2: Example Client Initial datagram without 0-RTT

A Client Initial packet exposes the version, source and destination connection IDs without encryption. The payload of the Initial packet is protected using the Initial secret. The complete TLS Client Hello, including any TLS Server Name Indication (SNI) present, is sent in one or more CRYPTO frames across one or more QUIC Initial packets.

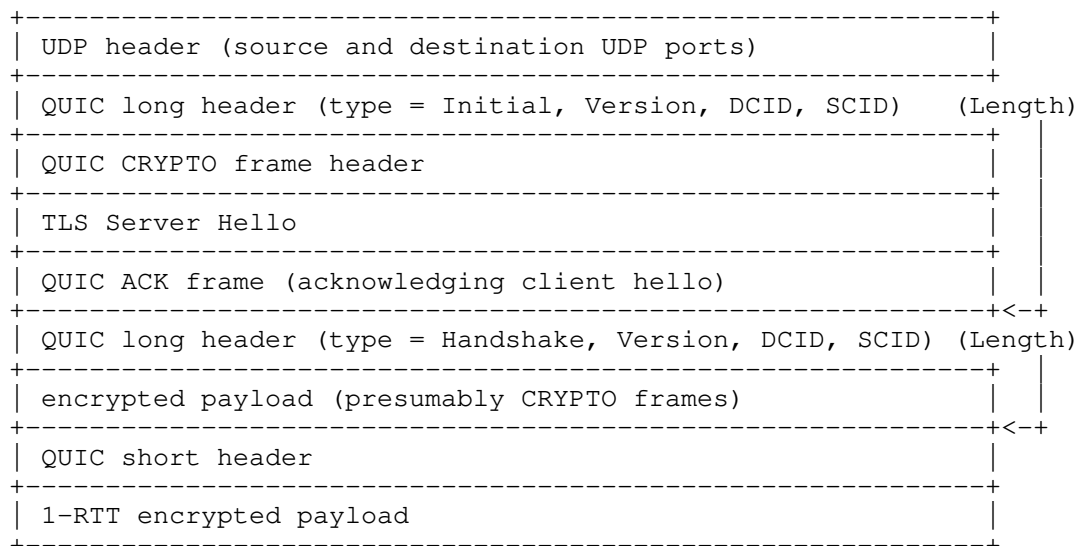


Figure 3: Coalesced Server Initial datagram pattern

The Server Initial datagram also exposes version number, source and destination connection IDs in the clear; the payload of the Initial packet(s) is protected using the Initial secret.

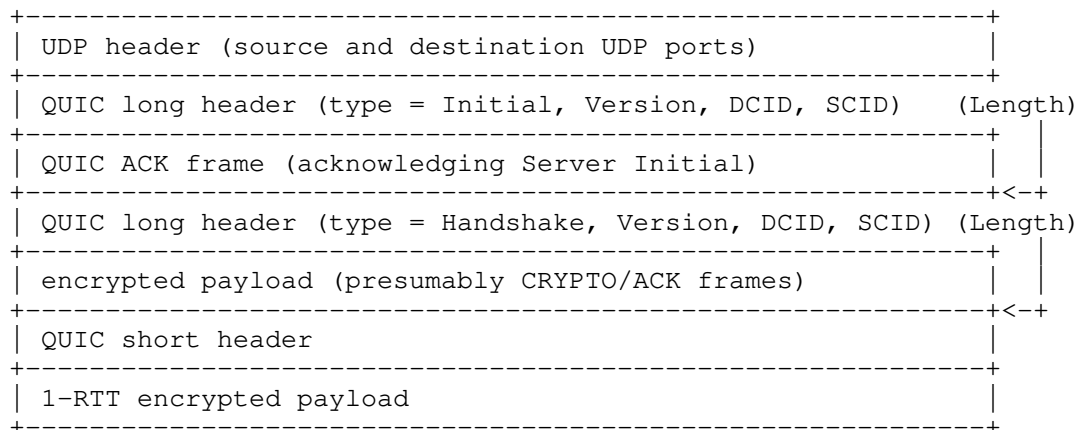


Figure 4: Coalesced Client Completion datagram pattern

The Client Completion flight does not expose any additional information; however, as the destination connection ID is server-selected, it usually is not the same ID that is sent in the Client Initial. Client Completion flights contain 1-RTT packets which indicate the handshake has completed (see Section 3.2) on the client, and for three-way handshake RTT estimation as in Section 3.8.

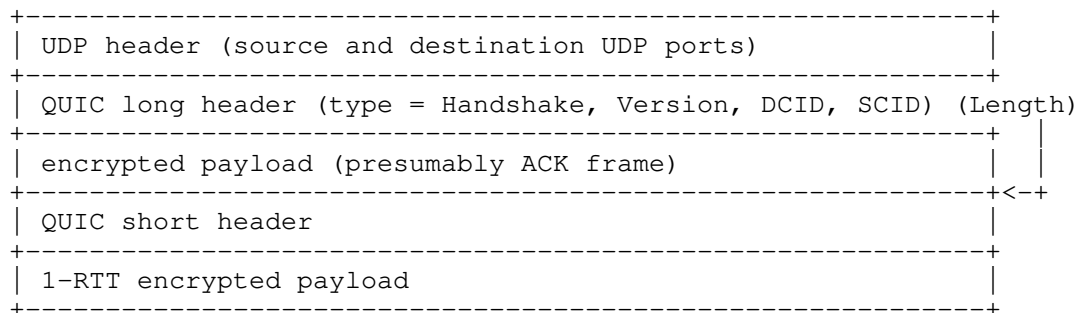


Figure 5: Coalesced Server Completion datagram pattern

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 data, the Client Initial flight can also include one or more 0-RTT packets, as shown in Figure 6.

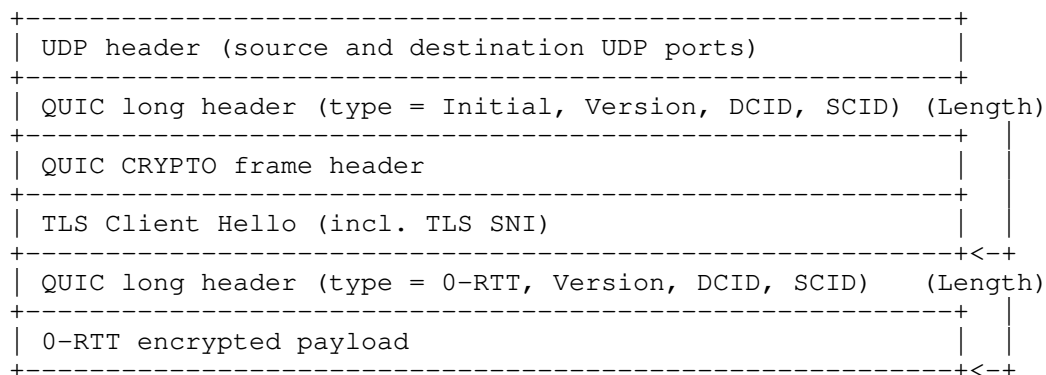


Figure 6: Coalesced 0-RTT Client Initial datagram

When a 0-RTT packet is coalesced with an Initial packet, the datagram will be padded to 1200 bytes. Additional datagrams containing only 0-RTT packets with long headers can be sent after the client Initial packet(s), containing more 0-RTT data. The amount of 0-RTT protected

data that can be sent in the first flight is limited by the initial congestion window, typically to 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.

2.6. Connection ID and Rebinding

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

Client and server each choose a connection ID during the handshake; for example, a server might request that a client use a connection ID, whereas the client might choose a zero-length value. 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 necessarily 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 that 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 Initial packets -- which are 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 could 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 an Initial packet with a reserved version number to trigger version negotiation. In the Version Negotiation packet, the connection IDs of the client's 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 on the Internet more often than with other commonly deployed Internet- and transport-layer protocols. Use of the version number field for traffic recognition will therefore behave differently than with these 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. Reliance on the version number field for the purposes of admission control is similarly likely to rapidly lead to unintended failure modes. Admission of QUIC traffic regardless of version avoids these failure modes, avoids unnecessary deployment delays, and supports continuous version-based evolution.

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, but typically without access to any keying information.

3.1. Identifying QUIC Traffic

The QUIC wire image is not specifically designed to be distinguishable from other UDP traffic by a passive observer in the network. While certain QUIC applications may be heuristically identifiable on a per-application basis, there is no general method for distinguishing QUIC traffic from otherwise-unclassifiable UDP traffic on a given link. Any unrecognized UDP traffic may therefore be QUIC traffic.

At the time of writing, two application bindings for QUIC have been published or adopted by the IETF: HTTP/3 [QUIC-HTTP] and DNS over Dedicated QUIC Connections [I-D.ietf-dprive-dnssoquic]. These are both known at the time of writing to have active Internet deployments, so an assumption that all QUIC traffic is HTTP/3 is not valid. HTTP/3 uses UDP port 443 by convention but various methods can be used to 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 [QUIC-GREASE].

Even though transport parameters transmitted in the client's Initial packet are observable by the network, they cannot be modified by the network without causing 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 might 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 Connection Confirmation 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 packets may be reordered or lost 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 packets that carry only acknowledgment (ACK-only) information from packets carrying upper-layer data in order to attempt to enhance performance, for example by queueing ACKs differently or manipulating ACK signaling [RFC3449]. Distinguishing ACK packets is possible in TCP, but is 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, SNI can be derived from the client's Initial packet(s) by calculating the Initial secret to decrypt the packet payload and parsing the QUIC CRYPTO frame(s) 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 (the second through fifth 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. However, it is expected that these versions will gradually disappear over time and therefore do not require any special consideration or treatment.

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 from the packet. The Initial secret is calculated 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.

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.

To determine the end of the packet 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 eighth bits of the header as described in Section 17.2 of [QUIC-TRANSPORT], but is protected 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(s) can be parsed to detect the CRYPTO frame(s) that contains the TLS ClientHello, which then can be parsed similarly to TLS over TCP connections. Note that there can be multiple CRYPTO frames spread out over one or more Initial packets, and they might not be in order, so reassembling the CRYPTO stream by parsing offsets and lengths is required. Further, the client's Initial packet(s) may contain other frames, so the first bytes of each frame need to be checked to identify the frame type and determine whether the frame can be skipped over. Note that the length of the frames is dependent on the frame type; see Section 18

of [QUIC-TRANSPORT]. E.g., PADDING frames, each consisting of a single zero byte, may occur before, after, or between CRYPTO frames. However, extensions might define additional frame types. If an unknown frame type is encountered, it is impossible to know the length of that frame which prevents skipping over it, and therefore parsing fails.

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. 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's address unintentionally changes, 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 5-tuple but different connection IDs might or might not belong to the same connection. Likewise, packets with the same connection ID but different 5-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. Note that QUIC packets containing only control frames (such as ACK-only packets) may be padded. Padding, though optional, may conceal connection roles or flow symmetry information.

3.8. Round-Trip Time (RTT) Measurement

The round-trip time (RTT) 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 all transport- and application-layer delay at both endpoints.

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, the latency spin bit in each direction changes value once per RTT any time that both endpoints are sending packets continuously. 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 all transport protocol delay (e.g., delayed sending of acknowledgments) 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 one-way packet delay variance as seen by an application end-point).

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. Limited observation of upstream congestion may be possible via the observation of CE markings in the IP header [RFC3168] on ECN-enabled QUIC traffic.

On-path devices can also make measurements of RTT, loss and other performance metrics when information is carried in an additional network-layer packet header (Section 6 of [I-D.ietf-tsvwg-transport-encrypt] describes use of operations, administration and management (OAM) information). Using network-layer approaches also has the advantage that common observation and analysis tools can be consistently used for multiple transport protocols, however, these techniques are often limited to measurements within one or multiple cooperating domains.

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-flow 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 network state timeout that is not less than 2 minutes for most UDP traffic. However, in practice, a QUIC endpoint can experience lower timeouts, in the range of 30 to 60 seconds [QUIC-TIMEOUT].

In contrast, [RFC5382] recommends a state 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 to 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.

The recommendation is therefore that, even when lower state timeouts are used for other UDP traffic, a state timeout of at least two minutes ought to be used 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 unnecessarily 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. In particular, the use of the connection ID for functions that require state to make a forwarding decision 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 TCP fallback. 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 a connection to survive client 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 address 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 8 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.

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 in other scenarios where packet filtering is desired. Though the guidance there holds, a particularly unwise behavior admits a handful of UDP packets and then makes a decision to whether or not filter later packets in the same connection. QUIC applications are encouraged to fall back to TCP if early packets do not arrive at their destination [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, Throttling, and NAT Binding

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 indiscriminately. When the handshake is blocked, QUIC-capable applications may fall back to TCP. However, blocking a random fraction of QUIC packets across 4-tuples will allow many QUIC handshakes to complete, preventing TCP fallback, but these 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.

Note that some source ports are assumed to be reflection attack vectors by some servers; see Section 8.1 of [QUIC-APPLICABILITY]. As a result, NAT binding to these source ports can result in that traffic being blocked.

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 0-RTT data 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 synccookies [RFC4937] are a well-established method of mitigating some kinds of TCP DDoS attacks. QUIC Retry packets are the functional analogue to synccookies, 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 Routing

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

Note that the 5-tuple of a QUIC connection can change due to migration. In this case different flows are observed by the path and maybe be treated differently, as congestion control is usually reset on migration (see also Section 3.5).

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 network segments support 1500-byte packets, but can only do so by fragmenting at a lower layer before traversing a network segment with a smaller MTU, and then reassembling within the network segment. This is permissible even when the IP layer is IPv6 or IPv4 with the DF bit set, because fragmentation 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, to avoid exceeding limited reassembly capacity.

For TCP, MSS clamping (Section 3.2 of [RFC4459]) is often used to change the sender's TCP maximum 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 encourages senders to approach the maximum packet size, which could then cause fragmentation within a network segment of which they may not be aware.

If path performance is limited when forwarding 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.

Networks with configurations that would lead to fragmentation of large packets within a network segment should drop such packets rather than fragmenting them. Network operators who plan to implement a more selective policy may start by focusing 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 networks supporting QUIC, it is recommended that a path drops any packet larger than the fragmentation size. When a QUIC endpoint uses DPLPMTUD, it will use a QUIC probe packet to discover the PMTU. If this probe is lost, it will 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].

A network cannot know in advance which discovery method is used by a QUIC endpoint, so it should send a PTB message in addition to dropping an 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 for PMTU discovery. This provides a signal to the endpoint to prevent the packet size from growing too large, which can entirely avoid network segment fragmentation for that flow.

Endpoints can cache PMTU information, in the IP-layer cache. This short-term consistency between the PMTU for flows can help avoid an endpoint using a PMTU that is inefficient. The IP cache can also influence the PMTU value of other IP flows that use the same path [RFC8201][DPLPMTUD], including IP packets carrying protocols other than QUIC. The representation of an IP path is implementation-specific [RFC8201].

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 required 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 significant text to and/or feedback on this document:

- * Chris Box
- * Dan Druta
- * David Schinazi
- * Gorrry Fairhurst

- * Ian Swett
- * Igor Lubashev
- * Jana Iyengar
- * Jared Mauch
- * Lars Eggert
- * Lucas Purdue
- * Marcus Ihlar
- * Mark Nottingham
- * Martin Duke
- * Martin Thomson
- * Matt Joras
- * Mike Bishop
- * Nick Banks
- * Thomas Fossati
- * Sean Turner

8. Acknowledgments

Special thanks to last call reviewers Elwyn Davies, Barry Lieba, Al Morton, and Peter Saint-Andre.

This work was 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., Ed. and S. Turner, Ed., "Using TLS to Secure QUIC", RFC 9001, DOI 10.17487/RFC9001, May 2021, <<https://www.rfc-editor.org/rfc/rfc9001>>.

[QUIC-TRANSPORT]

Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <<https://www.rfc-editor.org/rfc/rfc9000>>.

9.2. Informative References

[DOTS-ARCH]

Mortensen, A., Ed., Reddy, K., T., Ed., Andreasen, F., Teague, N., and R. Compton, "DDoS Open Threat Signaling (DOTS) Architecture", RFC 8811, DOI 10.17487/RFC8811, August 2020, <<https://www.rfc-editor.org/rfc/rfc8811>>.

[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-dprive-dnsquic]

Huitema, C., Dickinson, S., and A. Mankin, "DNS over Dedicated QUIC Connections", Work in Progress, Internet-Draft, draft-ietf-dprive-dnsquic-11, 21 March 2022, <<https://datatracker.ietf.org/doc/html/draft-ietf-dprive-dnsquic-11>>.

[I-D.ietf-tsvwg-transport-encrypt]

Fairhurst, G. and C. Perkins, "Considerations around Transport Header Confidentiality, Network Operations, and the Evolution of Internet Transport Protocols", Work in Progress, Internet-Draft, draft-ietf-tsvwg-transport-encrypt-21, 20 April 2021, <<https://datatracker.ietf.org/doc/html/draft-ietf-tsvwg-transport-encrypt-21>>.

[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-15, 7 March 2022, <<https://datatracker.ietf.org/doc/html/draft-ietf-quic-applicability-15>>.

[QUIC-GREASE]

Thomson, M., "Greasing the QUIC Bit", Work in Progress, Internet-Draft, draft-ietf-quic-bit-grease-02, 10 November 2021, <<https://datatracker.ietf.org/doc/html/draft-ietf-quic-bit-grease-02>>.

[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://datatracker.ietf.org/doc/html/draft-ietf-quic-http-34>>.

[QUIC-INVARIANTS]

Thomson, M., "Version-Independent Properties of QUIC", RFC 8999, DOI 10.17487/RFC8999, May 2021, <<https://www.rfc-editor.org/rfc/rfc8999>>.

[QUIC-RECOVERY]

Iyengar, J., Ed. and I. Swett, Ed., "QUIC Loss Detection and Congestion Control", RFC 9002, DOI 10.17487/RFC9002, May 2021, <<https://www.rfc-editor.org/rfc/rfc9002>>.

[QUIC-TIMEOUT]

Roskind, J., "QUIC (IETF-88 TSV Area Presentation)", 7 November 2013, <<https://www.ietf.org/proceedings/88/slides/slides-88-tsvarea-10.pdf>>.

[QUIC_LB]

Duke, M., Banks, N., and C. Huitema, "QUIC-LB: Generating Routable QUIC Connection IDs", Work in Progress, Internet-Draft, draft-ietf-quic-load-balancers-13, 28 March 2022, <<https://datatracker.ietf.org/doc/html/draft-ietf-quic-load-balancers-13>>.

[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>>.

- [RFC3168] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, DOI 10.17487/RFC3168, September 2001, <<https://www.rfc-editor.org/rfc/rfc3168>>.
- [RFC3449] Balakrishnan, H., Padmanabhan, V., Fairhurst, G., and M. Sooriyabandara, "TCP Performance Implications of Network Path Asymmetry", BCP 69, RFC 3449, DOI 10.17487/RFC3449, December 2002, <<https://www.rfc-editor.org/rfc/rfc3449>>.
- [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>>.

- [RFC7801] Dolmatov, V., Ed., "GOST R 34.12-2015: Block Cipher "Kuznyechik"", RFC 7801, DOI 10.17487/RFC7801, March 2016, <<https://www.rfc-editor.org/rfc/rfc7801>>.
- [RFC7838] Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", RFC 7838, DOI 10.17487/RFC7838, April 2016, <<https://www.rfc-editor.org/rfc/rfc7838>>.
- [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>>.
- [RFC9065] Fairhurst, G. and C. Perkins, "Considerations around Transport Header Confidentiality, Network Operations, and the Evolution of Internet Transport Protocols", RFC 9065, DOI 10.17487/RFC9065, July 2021, <<https://www.rfc-editor.org/rfc/rfc9065>>.
- [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-14, 13 February 2022, <<https://datatracker.ietf.org/doc/html/draft-ietf-tls-esni-14>>.
- [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>>.

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
Internet-Draft
Intended status: Standards Track
Expires: 8 September 2022

R. Marx
KU Leuven
L. Niccolini, Ed.
Facebook
M. Seemann, Ed.
Protocol Labs
7 March 2022

HTTP/3 and QPACK qlog event definitions
draft-ietf-quic-qlog-h3-events-01

Abstract

This document describes concrete qlog event definitions and their metadata for HTTP/3 and QPACK-related events. These events can then be embedded in the higher level schema defined in [QLOG-MAIN].

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 September 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction	3
1.1. Notational Conventions	3
2. Overview	4
2.1. Usage with QUIC	4
2.2. Links to the main schema	4
2.2.1. Raw packet and frame information	4
3. HTTP/3 and QPACK event definitions	5
3.1. http	5
3.1.1. parameters_set	5
3.1.2. parameters_restored	6
3.1.3. stream_type_set	7
3.1.4. frame_created	8
3.1.5. frame_parsed	8
3.1.6. push_resolved	9
3.2. qpack	9
3.2.1. state_updated	10
3.2.2. stream_state_updated	10
3.2.3. dynamic_table_updated	11
3.2.4. headers_encoded	11
3.2.5. headers_decoded	12
3.2.6. instruction_created	12
3.2.7. instruction_parsed	13
4. Security Considerations	13
5. IANA Considerations	13
6. References	13
6.1. Normative References	13
6.2. Informative References	14
Appendix A. HTTP/3 data field definitions	14
A.1. ProtocolEventBody extension	14
A.2. Owner	15
A.3. HTTP/3 Frames	15
A.3.1. DataFrame	15
A.3.2. HeadersFrame	15
A.3.3. CancelPushFrame	16
A.3.4. SettingsFrame	16
A.3.5. PushPromiseFrame	17
A.3.6. GoAwayFrame	17
A.3.7. MaxPushIDFrame	17
A.3.8. ReservedFrame	17
A.3.9. UnknownFrame	18
A.4. ApplicationError	18
Appendix B. QPACK DATA type definitions	18
B.1. ProtocolEventBody extension	18
B.2. QPACK Instructions	19
B.2.1. SetDynamicTableCapacityInstruction	19
B.2.2. InsertWithNameReferenceInstruction	19

B.2.3.	InsertWithoutNameReferenceInstruction	20
B.2.4.	DuplicateInstruction	20
B.2.5.	SectionAcknowledgementInstruction	20
B.2.6.	StreamCancellationInstruction	20
B.2.7.	InsertCountIncrementInstruction	20
B.3.	QPACK Header compression	21
B.3.1.	IndexedHeaderField	21
B.3.2.	LiteralHeaderFieldWithName	21
B.3.3.	LiteralHeaderFieldWithoutName	22
B.3.4.	QPACKHeaderBlockPrefix	22
B.3.5.	QPACKTableType	23
Appendix C.	Change Log	23
C.1.	Since draft-ietf-quic-qlog-h3-events-00:	23
C.2.	Since draft-marx-qlog-event-definitions-quic-h3-02:	23
C.3.	Since draft-marx-qlog-event-definitions-quic-h3-01:	23
C.4.	Since draft-marx-qlog-event-definitions-quic-h3-00:	25
Appendix D.	Design Variations	25
Appendix E.	Acknowledgements	25
Authors' Addresses	25

1. Introduction

This document describes the values of the qlog name ("category" + "event") and "data" fields and their semantics for the HTTP/3 and QPACK protocols. This document is based on draft-34 of the HTTP/3 I-D [QUIC-HTTP] and draft-21 of the QPACK I-D [QUIC-QPACK]. QUIC events are defined in a separate document [QLOG-QUIC].

Feedback and discussion are welcome at <https://github.com/quicwg/qlog> (<https://github.com/quicwg/qlog>). Readers are advised to refer to the "editor's draft" at that URL for an up-to-date version of this document.

Concrete examples of integrations of this schema in various programming languages can be found at <https://github.com/quiclog/qlog/> (<https://github.com/quiclog/qlog/>).

1.1. Notational Conventions

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 [RFC2119].

The event and data structure definitions in this document are expressed in the Concise Data Definition Language [CDDL] and its extensions described in [QLOG-MAIN].

2. Overview

This document describes the values of the qlog "name" ("category" + "event") and "data" fields and their semantics for the HTTP/3 and QPACK protocols.

This document assumes the usage of the encompassing main qlog schema defined in [QLOG-MAIN]. Each subsection below defines a separate category (for example http, qpack) and each subsubsection is an event type (for example frame_created).

For each event type, its importance and data definition is laid out, often accompanied by possible values for the optional "trigger" field. For the definition and semantics of "importance" and "trigger", see the main schema document.

Most of the complex datastructures, enums and re-usable definitions are grouped together on the bottom of this document for clarity.

2.1. Usage with QUIC

The events described in this document can be used with or without logging the related QUIC events defined in [QLOG-QUIC]. If used with QUIC events, the QUIC document takes precedence in terms of recommended filenames and trace separation setups.

If used without QUIC events, it is recommended that the implementation assign a globally unique identifier to each HTTP/3 connection. This ID can then be used as the value of the qlog "group_id" field, as well as the qlog filename or file identifier, potentially suffixed by the vantagepoint type (For example, abcd1234_server.qlog would contain the server-side trace of the connection with GUID abcd1234).

2.2. Links to the main schema

This document re-uses all the fields defined in the main qlog schema (e.g., name, category, type, data, group_id, protocol_type, the time-related fields, importance, RawInfo, etc.).

One entry in the "protocol_type" qlog array field MUST be "HTTP3" if events from this document are included in a qlog trace.

2.2.1. Raw packet and frame information

This document re-uses the definition of the RawInfo data class from [QLOG-MAIN].

Note: As HTTP/3 does not use trailers in frames, each HTTP/3 frame header_length can be calculated as `header_length = RawInfo:length - RawInfo:payload_length`

Note: In some cases, the length fields are also explicitly reflected inside of frame headers. For example, all HTTP/3 frames include their explicit payload lengths in the frame header. In these cases, those fields are intentionally preserved in the event definitions. Even though this can lead to duplicate data when the full RawInfo is logged, it allows a more direct mapping of the HTTP/3 specifications to qlog, making it easier for users to interpret. In this case, both fields MUST have the same value.

3. HTTP/3 and QPACK event definitions

Each subheading in this section is a qlog event category, while each sub-subheading is a qlog event type.

For example, for the following two items, we have the category "http" and event type "parameters_set", resulting in a concatenated qlog "name" field value of "http:parameters_set".

3.1. http

Note: like all category values, the "http" category is written in lowercase.

3.1.1. parameters_set

Importance: Base

This event contains HTTP/3 and QPACK-level settings, mostly those received from the HTTP/3 SETTINGS frame. All these parameters are typically set once and never change. However, they are typically set at different times during the connection, so there can be several instances of this event with different fields set.

Note that some settings have two variations (one set locally, one requested by the remote peer). This is reflected in the "owner" field. As such, this field MUST be correct for all settings included a single event instance. If you need to log settings from two sides, you MUST emit two separate event instances.

Note: we use the CDDL unwrap operator (~) here to make HTTPParameters into a re-usable list of fields. The unwrap operator copies the fields from the referenced type into the target type directly, extending the target with the unwrapped fields. TODO: explain this better + provide reference and maybe an example.

Definition:

```
HTTPParametersSet = {  
    ? owner: Owner  
  
    ~HTTPParameters  
  
    ; qlog-specific  
    ; indicates whether this implementation waits for a SETTINGS  
    ; frame before processing requests  
    ? waits_for_settings: bool  
}  
  
HTTPParameters = {  
    ? max_header_list_size: uint64  
    ? max_table_capacity: uint64  
    ? blocked_streams_count: uint64  
  
    ; additional settings for grease and extensions  
    * text => uint64  
}
```

Figure 1: HTTPParametersSet definition

Note: enabling server push is not explicitly done in HTTP/3 by use of a setting or parameter. Instead, it is communicated by use of the MAX_PUSH_ID frame, which should be logged using the frame_created and frame_parsed events below.

Additionally, this event can contain any number of unspecified fields. This is to reflect setting of for example unknown (greased) settings or parameters of (proprietary) extensions.

3.1.2. parameters_restored

Importance: Base

When using QUIC 0-RTT, HTTP/3 clients are expected to remember and reuse the server's SETTINGS from the previous connection. This event is used to indicate which HTTP/3 settings were restored and to which values when utilizing 0-RTT.

Definition:


```
HTTPParametersRestored = {  
    ~HTTPParameters  
}
```

Figure 2: HTTPParametersRestored definition

Note that, like for `parameters_set` above, this event can contain any number of unspecified fields to allow for additional and custom settings.

3.1.3. `stream_type_set`

Importance: Base

Emitted when a stream's type becomes known. This is typically when a stream is opened and the stream's type indicator is sent or received.

Note: most of this information can also be inferred by looking at a stream's id, since id's are strictly partitioned at the QUIC level. Even so, this event has a "Base" importance because it helps a lot in debugging to have this information clearly spelled out.

Definition:

```
HTTPStreamTypeSet = {  
    ? owner: Owner  
    stream_id: uint64  
  
    ? old: HTTPStreamType  
    new: HTTPStreamType  
  
    ; only when new === "push"  
    ? associated_push_id: uint64  
}  
  
HTTPStreamType = "data" /  
                 "control" /  
                 "push" /  
                 "reserved" /  
                 "qpack_encode" /  
                 "qpack_decode"
```

Figure 3: HTTPStreamTypeSet definition

3.1.4. frame_created

Importance: Core

HTTP equivalent to the packet_sent event. This event is emitted when the HTTP/3 framing actually happens. Note: this is not necessarily the same as when the HTTP/3 data is passed on to the QUIC layer. For that, see the "data_moved" event in [QLOG-QUIC].

Definition:

```
HTTPFrameCreated = {  
    stream_id: uint64  
    ? length: uint64  
    frame: HTTPFrame  
    ? raw: RawInfo  
}
```

Figure 4: HTTPFrameCreated definition

Note: in HTTP/3, DATA frames can have arbitrarily large lengths to reduce frame header overhead. As such, DATA frames can span many QUIC packets and can be created in a streaming fashion. In this case, the frame_created event is emitted once for the frame header, and further streamed data is indicated using the data_moved event.

3.1.5. frame_parsed

Importance: Core

HTTP equivalent to the packet_received event. This event is emitted when we actually parse the HTTP/3 frame. Note: this is not necessarily the same as when the HTTP/3 data is actually received on the QUIC layer. For that, see the "data_moved" event in [QLOG-QUIC].

Definition:

```
HTTPFrameParsed = {  
    stream_id: uint64  
    ? length: uint64  
    frame: HTTPFrame  
    ? raw: RawInfo  
}
```

Figure 5: HTTPFrameParsed definition

Note: in HTTP/3, DATA frames can have arbitrarily large lengths to reduce frame header overhead. As such, DATA frames can span many QUIC packets and can be processed in a streaming fashion. In this case, the `frame_parsed` event is emitted once for the frame header, and further streamed data is indicated using the `data_moved` event.

3.1.6. `push_resolved`

Importance: Extra

This event is emitted when a pushed resource is successfully claimed (used) or, conversely, abandoned (rejected) by the application on top of HTTP/3 (e.g., the web browser). This event is added to help debug problems with unexpected PUSH behaviour, which is commonplace with HTTP/2.

Definition:

```
HTTPPushResolved = {  
    ? push_id: uint64  
  
    ; in case this is logged from a place that does not have access  
    ; to the push_id  
    ? stream_id: uint64  
  
    decision: HTTPPushDecision  
}  
  
HTTPPushDecision = "claimed" / "abandoned"
```

Figure 6: HTTPPushResolved definition

3.2. `qpack`

Note: like all category values, the `"qpack"` category is written in lowercase.

The QPACK events mainly serve as an aid to debug low-level QPACK issues. The higher-level, plaintext header values SHOULD (also) be logged in the `http.frame_created` and `http.frame_parsed` event data (instead).

Note: `qpack` does not have its own `parameters_set` event. This was merged with `http.parameters_set` for brevity, since `qpack` is a required extension for HTTP/3 anyway. Other HTTP/3 extensions MAY also log their `SETTINGS` fields in `http.parameters_set` or MAY define their own events.

3.2.1. state_updated

Importance: Base

This event is emitted when one or more of the internal QPACK variables changes value. Note that some variables have two variations (one set locally, one requested by the remote peer). This is reflected in the "owner" field. As such, this field MUST be correct for all variables included a single event instance. If you need to log settings from two sides, you MUST emit two separate event instances.

Definition:

```
QPACKStateUpdate = {  
  owner: Owner  
  ? dynamic_table_capacity: uint64  
  
  ; effective current size, sum of all the entries  
  ? dynamic_table_size: uint64  
  ? known_received_count: uint64  
  ? current_insert_count: uint64  
}
```

Figure 7: QPACKStateUpdate definition

3.2.2. stream_state_updated

Importance: Core

This event is emitted when a stream becomes blocked or unblocked by header decoding requests or QPACK instructions.

Note: This event is of "Core" importance, as it might have a large impact on HTTP/3's observed performance.

Definition:

```
QPACKStreamStateUpdate = {  
  stream_id: uint64  
  ; streams are assumed to start "unblocked"  
  ; until they become "blocked"  
  state: QPACKStreamState  
}
```

QPACKStreamState = "blocked" / "unblocked"

Figure 8: QPACKStreamStateUpdate definition

3.2.3. dynamic_table_updated

Importance: Extra

This event is emitted when one or more entries are inserted or evicted from QPACK's dynamic table.

Definition:

```
QPACKDynamicTableUpdate = {  
  ; local = the encoder's dynamic table  
  ; remote = the decoder's dynamic table  
  owner: Owner  
  
  update_type: QPACKDynamicTableUpdateType  
  entries: [+ QPACKDynamicTableEntry]  
}  
  
QPACKDynamicTableUpdateType = "inserted" / "evicted"  
  
QPACKDynamicTableEntry = {  
  index: uint64  
  ? name: text / hexstring  
  ? value: text / hexstring  
}
```

Figure 9: QPACKDynamicTableUpdate definition

3.2.4. headers_encoded

Importance: Base

This event is emitted when an uncompressed header block is encoded successfully.

Note: this event has overlap with http.frame_created for the HeadersFrame type. When outputting both events, implementers MAY omit the "headers" field in this event.

Definition:

```
QPACKHeadersEncoded = {  
  ? stream_id: uint64  
  ? headers: [+ HTTPField]  
  
  block_prefix: QPACKHeaderBlockPrefix  
  header_block: [+ QPACKHeaderBlockRepresentation]  
  
  ? length: uint  
  ? raw: hexstring  
}
```

Figure 10: QPACKHeadersEncoded definition

3.2.5. headers_decoded

Importance: Base

This event is emitted when a compressed header block is decoded successfully.

Note: this event has overlap with `http.frame_parsed` for the `HeadersFrame` type. When outputting both events, implementers MAY omit the "headers" field in this event.

Definition:

```
QPACKHeadersDecoded = {  
  ? stream_id: uint64  
  ? headers: [+ HTTPField]  
  
  block_prefix: QPACKHeaderBlockPrefix  
  header_block: [+ QPACKHeaderBlockRepresentation]  
  
  ? length: uint32  
  ? raw: hexstring  
}
```

Figure 11: QPACKHeadersDecoded definition

3.2.6. instruction_created

Importance: Base

This event is emitted when a QPACK instruction (both decoder and encoder) is created and added to the encoder/decoder stream.

Definition:

```
QPACKInstructionCreated = {  
    ; see definition in appendix  
    instruction: QPACKInstruction  
    ? length: uint32  
    ? raw: hexstring  
}
```

Figure 12: QPACKInstructionCreated definition

Note: encoder/decoder semantics and stream_id's are implicit in either the instruction types or can be logged via other events (e.g., http.stream_type_set)

3.2.7. instruction_parsed

Importance: Base

This event is emitted when a QPACK instruction (both decoder and encoder) is read from the encoder/decoder stream.

Definition:

```
QPACKInstructionParsed = {  
    ; see QPACKInstruction definition in appendix  
    instruction: QPACKInstruction  
  
    ? length: uint32  
    ? raw: hexstring  
}
```

Figure 13: QPACKInstructionParsed definition

Note: encoder/decoder semantics and stream_id's are implicit in either the instruction types or can be logged via other events (e.g., http.stream_type_set)

4. Security Considerations

TBD

5. IANA Considerations

TBD

6. References

6.1. Normative References

[CDDL] Birkholz, H., Vigano, C., and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures", RFC 8610, DOI 10.17487/RFC8610, June 2019, <<https://www.rfc-editor.org/rfc/rfc8610>>.

[QLOG-MAIN] Marx, R., Ed., Niccolini, L., Ed., and M. Seemann, Ed., "Main logging schema for qlog", Work in Progress, Internet-Draft, draft-ietf-quic-qlog-main-schema-02, <<https://datatracker.ietf.org/doc/html/draft-ietf-quic-qlog-main-schema-02>>.

[QLOG-QUIC] Marx, R., Ed., Niccolini, L., Ed., and M. Seemann, Ed., "QUIC event definitions for qlog", Work in Progress, Internet-Draft, draft-ietf-quic-qlog-quic-events-01, <<https://datatracker.ietf.org/doc/html/draft-ietf-quic-qlog-quic-events-01>>.

[QUIC-HTTP] Bishop, M., Ed., "Hypertext Transfer Protocol Version 3 (HTTP/3)", Work in Progress, Internet-Draft, draft-ietf-quic-http-latest, <<https://datatracker.ietf.org/doc/html/draft-ietf-quic-http-latest>>.

[QUIC-QPACK] Krasic, C., Bishop, M., and A. Frindell, Ed., "QPACK: Header Compression for HTTP over QUIC", Work in Progress, Internet-Draft, draft-ietf-quic-qpack-latest, <<https://datatracker.ietf.org/doc/html/draft-ietf-quic-qpack-latest>>.

6.2. Informative References

[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/rfc/rfc2119>>.

Appendix A. HTTP/3 data field definitions

A.1. ProtocolEventBody extension

We extend the \$ProtocolEventBody extension point defined in [QLOG-MAIN] with the HTTP/3 protocol events defined in this document.


```
HTTPEvents = HTTPParametersSet / HTTPParametersRestored /  
             HTTPStreamTypeSet / HTTPFrameCreated /  
             HTTPFrameParsed / HTTPPushResolved  
  
$ProtocolEventBody /= HTTPEvents
```

Figure 14: HTTPEvents definition and ProtocolEventBody extension

A.2. Owner

```
Owner = "local" / "remote"
```

Figure 15: Owner definition

A.3. HTTP/3 Frames

```
HTTPFrame = HTTPDataFrame /  
            HTTPHeadersFrame /  
            HTTPCancelPushFrame /  
            HTTPSettingsFrame /  
            HTTPPushPromiseFrame /  
            HTTPGoawayFrame /  
            HTTPMaxPushIDFrame /  
            HTTPReservedFrame /  
            UnknownFrame
```

Figure 16: HTTPFrame definition

A.3.1. DataFrame

```
HTTPDataFrame = {  
    frame_type: "data"  
    ? raw: hexstring  
}
```

Figure 17: HTTPDataFrame definition

A.3.2. HeadersFrame

This represents an `_uncompressed_`, plaintext HTTP Headers frame (e.g., no QPACK compression is applied).

For example:

```
headers: [  
  {  
    "name": ":path",  
    "value": "/"  
  },  
  {  
    "name": ":method",  
    "value": "GET"  
  },  
  {  
    "name": ":authority",  
    "value": "127.0.0.1:4433"  
  },  
  {  
    "name": ":scheme",  
    "value": "https"  
  }  
]
```

Figure 18: HTTPHeadersFrame example

```
HTTPHeadersFrame = {  
  frame_type: "headers"  
  headers: [* HTTPField]  
}
```

Figure 19: HTTPHeadersFrame definition

```
HTTPField = {  
  name: text  
  value: text  
}
```

Figure 20: HTTPField definition

A.3.3. CancelPushFrame

```
HTTPCancelPushFrame = {  
  frame_type: "cancel_push"  
  push_id: uint64  
}
```

Figure 21: HTTPCancelPushFrame definition

A.3.4. SettingsFrame

```
HTTPSettingsFrame = {  
    frame_type: "settings"  
    settings: [* HTTPSetting]  
}  
  
HTTPSetting = {  
    name: text  
    value: uint64  
}
```

Figure 22: HTTPSettingsFrame definition

A.3.5. PushPromiseFrame

```
HTTPPushPromiseFrame = {  
    frame_type: "push_promise"  
    push_id: uint64  
    headers: [* HTTPField]  
}
```

Figure 23: HTTPPushPromiseFrame definition

A.3.6. GoAwayFrame

```
HTTPGoawayFrame = {  
    frame_type: "goaway"  
  
    ; Either stream_id or push_id.  
    ; This is implicit from the sender of the frame  
    id: uint64  
}
```

Figure 24: HTTPGoawayFrame definition

A.3.7. MaxPushIDFrame

```
HTTPMaxPushIDFrame = {  
    frame_type: "max_push_id"  
    push_id: uint64  
}
```

Figure 25: HTTPMaxPushIDFrame definition

A.3.8. ReservedFrame

```

HTTPReservedFrame = {
    frame_type: "reserved"

    ? length: uint64
}

```

Figure 26: HTTPReservedFrame definition

A.3.9. UnknownFrame

HTTP/3 qlog re-uses QUIC's UnknownFrame definition, since their values and usage overlaps. See [QLOG-QUIC].

A.4. ApplicationError

```

HTTPApplicationError = "http_no_error" /
                        "http_general_protocol_error" /
                        "http_internal_error" /
                        "http_stream_creation_error" /
                        "http_closed_critical_stream" /
                        "http_frame_unexpected" /
                        "http_frame_error" /
                        "http_excessive_load" /
                        "http_id_error" /
                        "http_settings_error" /
                        "http_missing_settings" /
                        "http_request_rejected" /
                        "http_request_cancelled" /
                        "http_request_incomplete" /
                        "http_early_response" /
                        "http_connect_error" /
                        "http_version_fallback"

```

Figure 27: HTTPApplicationError definition

The HTTPApplicationError defines the general \$ApplicationError definition in the qlog QUIC definition, see [QLOG-QUIC].

```

; ensure HTTP errors are properly validate in QUIC events as well
; e.g., QUIC's ConnectionClose Frame
$ApplicationError /= HTTPApplicationError

```

Appendix B. QPACK DATA type definitions

B.1. ProtocolEventBody extension

We extend the \$ProtocolEventBody extension point defined in [QLOG-MAIN] with the QPACK protocol events defined in this document.

```
QPACKEvents = QPACKStateUpdate / QPACKStreamStateUpdate /
              QPACKDynamicTableUpdate / QPACKHeadersEncoded /
              QPACKHeadersDecoded / QPACKInstructionCreated /
              QPACKInstructionParsed
```

```
$ProtocolEventBody /= QPACKEvents
```

Figure 28: QPACKEvents definition and ProtocolEventBody extension

B.2. QPACK Instructions

Note: the instructions do not have explicit encoder/decoder types, since there is no overlap between the instructions of both types in neither name nor function.

```
QPACKInstruction = SetDynamicTableCapacityInstruction /
                  InsertWithNameReferenceInstruction /
                  InsertWithoutNameReferenceInstruction /
                  DuplicateInstruction /
                  SectionAcknowledgementInstruction /
                  StreamCancellationInstruction /
                  InsertCountIncrementInstruction
```

Figure 29: QPACKInstruction definition

B.2.1. SetDynamicTableCapacityInstruction

```
SetDynamicTableCapacityInstruction = {
  instruction_type: "set_dynamic_table_capacity"
  capacity: uint32
}
```

Figure 30: SetDynamicTableCapacityInstruction definition

B.2.2. InsertWithNameReferenceInstruction

```
InsertWithNameReferenceInstruction = {
  instruction_type: "insert_with_name_reference"
  table_type: QPACKTableType
  name_index: uint32
  huffman_encoded_value: bool
  ? value_length: uint32
  ? value: text
}
```

Figure 31: InsertWithNameReferenceInstruction definition

B.2.3. InsertWithoutNameReferenceInstruction

```
InsertWithoutNameReferenceInstruction = {  
  instruction_type: "insert_without_name_reference"  
  huffman_encoded_name: bool  
  ? name_length: uint32  
  ? name: text  
  huffman_encoded_value: bool  
  ? value_length: uint32  
  ? value: text  
}
```

Figure 32: InsertWithoutNameReferenceInstruction definition

B.2.4. DuplicateInstruction

```
DuplicateInstruction = {  
  instruction_type: "duplicate"  
  index: uint32  
}
```

Figure 33: DuplicateInstruction definition

B.2.5. SectionAcknowledgementInstruction

```
SectionAcknowledgementInstruction = {  
  instruction_type: "section_acknowledgement"  
  stream_id: uint64  
}
```

Figure 34: SectionAcknowledgementInstruction definition

B.2.6. StreamCancellationInstruction

```
StreamCancellationInstruction = {  
  instruction_type: "stream_cancellation"  
  stream_id: uint64  
}
```

Figure 35: StreamCancellationInstruction definition

B.2.7. InsertCountIncrementInstruction

```
InsertCountIncrementInstruction = {  
  instruction_type: "insert_count_increment"  
  increment: uint32  
}
```

Figure 36: InsertCountIncrementInstruction definition

B.3. QPACK Header compression

```
QPACKHeaderBlockRepresentation = IndexedHeaderField /  
                                LiteralHeaderFieldWithName /  
                                LiteralHeaderFieldWithoutName
```

Figure 37: QPACKHeaderBlockRepresentation definition

B.3.1. IndexedHeaderField

Note: also used for "indexed header field with post-base index"

```
IndexedHeaderField = {  
    header_field_type: "indexed_header"  
  
    ; MUST be "dynamic" if is_post_base is true  
    table_type: QPACKTableType  
    index: uint32  
  
    ; to represent the "indexed header field with post-base index"  
    ; header field type  
    is_post_base: bool .default false  
}
```

Figure 38: IndexedHeaderField definition

B.3.2. LiteralHeaderFieldWithName

Note: also used for "Literal header field with post-base name reference"

```
LiteralHeaderFieldWithName = {  
    header_field_type: "literal_with_name"  
  
    ; the 3rd "N" bit  
    preserve_literal: bool  
  
    ; MUST be "dynamic" if is_post_base is true  
    table_type: QPACKTableType  
    name_index: uint32  
    huffman_encoded_value: bool  
    ? value_length: uint32  
    ? value: text  
  
    ; to represent the "indexed header field with post-base index"  
    ; header field type  
    is_post_base: bool .default false  
}
```

Figure 39: LiteralHeaderFieldWithName definition

B.3.3. LiteralHeaderFieldWithoutName

```
LiteralHeaderFieldWithoutName = {  
    header_field_type: "literal_without_name"  
  
    ; the 3rd "N" bit  
    preserve_literal: bool  
    huffman_encoded_name: bool  
    ? name_length: uint32  
    ? name: text  
  
    huffman_encoded_value: bool  
    ? value_length: uint32  
    ? value: text  
}
```

Figure 40: LiteralHeaderFieldWithoutName definition

B.3.4. QPACKHeaderBlockPrefix

```
QPACKHeaderBlockPrefix = {  
    required_insert_count: uint32  
    sign_bit: bool  
    delta_base: uint32  
}
```

Figure 41: QPACKHeaderBlockPrefix definition

B.3.5. QPACKTableType

```
QPACKTableType = "static" / "dynamic"
```

Figure 42: QPACKTableType definition

Appendix C. Change Log

C.1. Since draft-ietf-quic-qlog-h3-events-00:

- * Change the data definition language from TypeScript to CDDL (#143)

C.2. Since draft-marx-qlog-event-definitions-quic-h3-02:

- * These changes were done in preparation of the adoption of the drafts by the QUIC working group (#137)
- * Split QUIC and HTTP/3 events into two separate documents
- * Moved RawInfo, Importance, Generic events and Simulation events to the main schema document.

C.3. Since draft-marx-qlog-event-definitions-quic-h3-01:

Major changes:

- * Moved data_moved from http to transport. Also made the "from" and "to" fields flexible strings instead of an enum (#111,#65)
- * Moved packet_type fields to PacketHeader. Moved packet_size field out of PacketHeader to RawInfo:length (#40)
- * Made events that need to log packet_type and packet_number use a header field instead of logging these fields individually
- * Added support for logging retry, stateless reset and initial tokens (#94,#86,#117)
- * Moved separate general event categories into a single category "generic" (#47)
- * Added "transport:connection_closed" event (#43,#85,#78,#49)
- * Added version_information and alpn_information events (#85,#75,#28)
- * Added parameters_restored events to help clarify 0-RTT behaviour (#88)

Smaller changes:

- * Merged loss_timer events into one loss_timer_updated event
- * Field data types are now strongly defined (#10, #39, #36, #115)
- * Renamed qpack instruction_received and instruction_sent to instruction_created and instruction_parsed (#114)
- * Updated qpack:dynamic_table_updated.update_type. It now has the value "inserted" instead of "added" (#113)
- * Updated qpack:dynamic_table_updated. It now has an "owner" field to differentiate encoder vs decoder state (#112)
- * Removed push_allowed from http:parameters_set (#110)
- * Removed explicit trigger field indications from events, since this was moved to be a generic property of the "data" field (#80)
- * Updated transport:connection_id_updated to be more in line with other similar events. Also dropped importance from Core to Base (#45)
- * Added length property to PaddingFrame (#34)
- * Added packet_number field to transport:frames_processed (#74)
- * Added a way to generically log packet header flags (first 8 bits) to PacketHeader
- * Added additional guidance on which events to log in which situations (#53)
- * Added "simulation:scenario" event to help indicate simulation details
- * Added "packets_acked" event (#107)
- * Added "datagram_ids" to the datagram_X and packet_X events to allow tracking of coalesced QUIC packets (#91)
- * Extended connection_state_updated with more fine-grained states (#49)

C.4. Since draft-marx-qlog-event-definitions-quic-h3-00:

- * Event and category names are now all lowercase
- * Added many new events and their definitions
- * "type" fields have been made more specific (especially important for PacketType fields, which are now called packet_type instead of type)
- * Events are given an importance indicator (issue #22)
- * Event names are more consistent and use past tense (issue #21)
- * Triggers have been redefined as properties of the "data" field and updated for most events (issue #23)

Appendix D. Design Variations

TBD

Appendix E. Acknowledgements

Much of the initial work by Robin Marx was done at Hasselt University.

Thanks to Marten Seemann, Jana Iyengar, Brian Trammell, Dmitri Tikhonov, Stephen Petrides, Jari Arkko, Marcus Ihlar, Victor Vasiliev, Mirja Kuehlewind, Jeremy Laine, Kazu Yamamoto, Christian Huitema, and Lucas Pardue for their feedback and suggestions.

Authors' Addresses

Robin Marx
KU Leuven
Email: robin.marx@kuleuven.be

Luca Niccolini (editor)
Facebook
Email: lniccolini@fb.com

Marten Seemann (editor)
Protocol Labs
Email: marten@protocol.ai

QUIC
Internet-Draft
Intended status: Standards Track
Expires: 8 September 2022

R. Marx, Ed.
KU Leuven
L. Niccolini, Ed.
Facebook
M. Seemann, Ed.
Protocol Labs
7 March 2022

Main logging schema for qlog
draft-ietf-quic-qlog-main-schema-02

Abstract

This document describes a high-level schema for a standardized logging format called qlog. This format allows easy sharing of data and the creation of reusable visualization and debugging tools. The high-level schema in this document is intended to be protocol-agnostic. Separate documents specify how the format should be used for specific protocol data. The schema is also format-agnostic, and can be represented in for example JSON, csv or protobuf.

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 September 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights

and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction	3
1.1. Notational Conventions	4
1.1.1. Schema definition	4
1.1.2. Serialization	5
2. Design goals	6
3. The high level qlog schema	6
3.1. Summary	7
3.2. traces	8
3.3. Individual Trace containers	9
3.3.1. Configuration	10
3.3.2. vantage_point	12
3.4. Field name semantics	13
3.4.1. Timestamps	15
3.4.2. Category and Event Type	16
3.4.3. Data	17
3.4.4. protocol_type	19
3.4.5. Triggers	19
3.4.6. group_id	20
3.4.7. common_fields	21
4. Guidelines for event definition documents	23
4.1. Event design guidelines	24
4.2. Event importance indicators	24
4.3. Custom fields	25
5. Generic events and data classes	26
5.1. Raw packet and frame information	26
5.2. Generic events	27
5.2.1. error	27
5.2.2. warning	28
5.2.3. info	28
5.2.4. debug	28
5.2.5. verbose	29
5.3. Simulation events	29
5.3.1. scenario	29
5.3.2. marker	30
6. Serializing qlog	30
6.1. qlog to JSON mapping	31
6.1.1. I-JSON	31
6.1.2. Truncated values	32
6.2. qlog to JSON Text Sequences mapping	33
6.2.1. Supporting JSON Text Sequences in tooling	36
6.3. Other optimized formatting options	36

6.3.1. Data structure optimizations	37
6.3.2. Compression	38
6.3.3. Binary formats	39
6.3.4. Overview and summary	40
6.4. Conversion between formats	41
7. Methods of access and generation	42
7.1. Set file output destination via an environment variable	42
7.2. Access logs via a well-known endpoint	44
8. Tooling requirements	44
9. Security and privacy considerations	45
10. IANA Considerations	45
11. References	45
11.1. Normative References	45
11.2. Informative References	47
Appendix A. Change Log	47
A.1. Since draft-ietf-quic-qlog-main-schema-01:	47
A.2. Since draft-ietf-quic-qlog-main-schema-00:	47
A.3. Since draft-marx-qlog-main-schema-draft-02:	47
A.4. Since draft-marx-qlog-main-schema-01:	48
A.5. Since draft-marx-qlog-main-schema-00:	48
Appendix B. Design Variations	48
Appendix C. Acknowledgements	49
Authors' Addresses	49

1. Introduction

There is currently a lack of an easily usable, standardized endpoint logging format. Especially for the use case of debugging and evaluating modern Web protocols and their performance, it is often difficult to obtain structured logs that provide adequate information for tasks like problem root cause analysis.

This document aims to provide a high-level schema and harness that describes the general layout of an easily usable, shareable, aggregatable and structured logging format. This high-level schema is protocol agnostic, with logging entries for specific protocols and use cases being defined in other documents (see for example [QLOG-QUIC] for QUIC and [QLOG-H3] for HTTP/3 and QPACK-related event definitions).

The goal of this high-level schema is to provide amenities and default characteristics that each logging file should contain (or should be able to contain), such that generic and reusable toolsets can be created that can deal with logs from a variety of different protocols and use cases.

As such, this document contains concepts such as versioning, metadata inclusion, log aggregation, event grouping and log file size reduction techniques.

Feedback and discussion are welcome at <https://github.com/quicwg/qlog> (<https://github.com/quicwg/qlog>). Readers are advised to refer to the "editor's draft" at that URL for an up-to-date version of this document.

Concrete examples of integrations of this schema in various programming languages can be found at <https://github.com/quiclog/qlog/> (<https://github.com/quiclog/qlog/>).

1.1. Notational Conventions

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 [RFC2119].

1.1.1. Schema definition

To define events and data structures, all qlog documents use the Concise Data Definition Language [CDDL]. This document uses the basic syntax, the specific text, uint, float32, float64, bool, and any types, as well as the .default, .size, and .regex control operators, the ~ unwrapping operator, and the \$ extension point syntax from [CDDL].

Additionally, this document defines the following custom types for clarity:

```
; CDDL's uint is defined as being 64-bit in size
; but for many protocol fields we want to be more restrictive
; and explicit
uint8 = uint .size 1
uint16 = uint .size 2
uint32 = uint .size 4
uint64 = uint .size 8

; an even-length lowercase string of hexadecimally encoded bytes
; examples: 82dc, 027339, 4cdbfd9bf0
; this is needed because the default CDDL binary string (bytes/bstr)
; is only CBOR and not JSON compatible
hexstring = text .regex "([0-9a-f]{2})*"
```

Figure 1: Additional CDDL type definitions

The main general CDDL syntax conventions in this document a reader should be aware of for easy reading comprehension are:

- * ? obj : this object is optional
- * TypeName1 / TypeName2 : a union of these two types (object can be either type 1 OR type 2)
- * obj: TypeName : this object has this concrete type
- * obj: [* TypeName] : this object is an array of this type with minimum size of 0 elements
- * obj: [+ TypeName] : this object is an array of this type with minimum size of 1 element
- * TypeName = ... : defines a new type
- * EnumName = "entry1" / "entry2" / entry3 / ...: defines an enum
- * StructName = { ... } : defines a new struct type
- * ; : single-line comment
- * * text => any : special syntax to indicate 0 or more fields that have a string key that maps to any value. Used to indicate a generic JSON object.

All timestamps and time-related values (e.g., offsets) in qlog are logged as float64 in the millisecond resolution.

Other qlog documents can define their own CDDL-compatible (struct) types (e.g., separately for each Packet type that a protocol supports).

1.1.2. Serialization

While the qlog schemas are format-agnostic, and can be serialized in many ways (e.g., JSON, CBOR, protobuf, ...), this document only describes how to employ [JSON], its subset [I-JSON], and its streamable derivative [JSON-Text-Sequences] as textual serialization options. As such, examples are provided in [JSON]. Other documents may describe how to utilize other concrete serialization options, though tips and requirements for these are also listed in this document (Section 6).

2. Design goals

The main tenets for the qlog schema design are:

- * Streamable, event-based logging
- * Flexibility in the format, complexity in the tooling (e.g., few components are a MUST, tools need to deal with this)
- * Extensible and pragmatic
- * Aggregation and transformation friendly (e.g., the top-level element for the non-streaming format is a container for individual traces, group_ids can be used to tag events to a particular context)
- * Metadata is stored together with event data

3. The high level qlog schema

A qlog file should be able to contain several individual traces and logs from multiple vantage points that are in some way related. To that end, the top-level element in the qlog schema defines only a small set of "header" fields and an array of component traces. For this document, the required "qlog_version" field MUST have a value of "0.3".

Note: there have been several previously broadly deployed qlog versions based on older drafts of this document (see draft-marx-qlog-main-schema). The old values for the "qlog_version" field were "draft-00", "draft-01" and "draft-02". When qlog was moved to the QUIC working group, we decided to switch to a new versioning scheme which is independent of individual draft document numbers. However, we did start from 0.3, as conceptually 0.0, 0.1 and 0.2 can map to draft-00, draft-01 and draft-02.

As qlog can be serialized in a variety of ways, the "qlog_format" field is used to indicate which serialization option was chosen. Its value MUST either be one of the options defined in this document (e.g., Section 6) or the field must be omitted entirely, in which case it assumes the default value of "JSON".

In order to make it easier to parse and identify qlog files and their serialization format, the "qlog_version" and "qlog_format" fields and their values SHOULD be in the first 256 characters/bytes of the resulting log file.

An example of the qlog file's top-level structure is shown in Figure 2.

Definition:

```
QlogFile = {  
  qlog_version: text  
  ? qlog_format: text .default "JSON"  
  ? title: text  
  ? description: text  
  ? summary: Summary  
  ? traces: [+ Trace / TraceError]  
}
```

Figure 2: QlogFile definition

JSON serialization example:

```
{  
  "qlog_version": "0.3",  
  "qlog_format": "JSON",  
  "title": "Name of this particular qlog file (short)",  
  "description": "Description for this group of traces (long)",  
  "summary": {  
    ...  
  },  
  "traces": [...]  
}
```

Figure 3: QlogFile example

3.1. Summary

In a real-life deployment with a large amount of generated logs, it can be useful to sort and filter logs based on some basic summarized or aggregated data (e.g., log length, packet loss rate, log location, presence of error events, ...). The summary field (if present) SHOULD be on top of the qlog file, as this allows for the file to be processed in a streaming fashion (i.e., the implementation could just read up to and including the summary field and then only load the full logs that are deemed interesting by the user).

As the summary field is highly deployment-specific, this document does not specify any default fields or their semantics. Some examples of potential entries are shown in Section 3.1.

Definition:

```
Summary = {  
    ; summary can contain any type of custom information  
    ; text here doesn't mean the type text,  
    ; but the fact that keys/names in the objects are strings  
    * text => any  
}
```

Figure 4: Summary definition

JSON serialization example:

```
{  
    "trace_count": 1,  
    "max_duration": 5006,  
    "max_outgoing_loss_rate": 0.013,  
    "total_event_count": 568,  
    "error_count": 2  
}
```

Figure 5: Summary example

3.2. traces

It is often advantageous to group several related qlog traces together in a single file. For example, we can simultaneously perform logging on the client, on the server and on a single point on their common network path. For analysis, it is useful to aggregate these three individual traces together into a single file, so it can be uniquely stored, transferred and annotated.

As such, the "traces" array contains a list of individual qlog traces. Typical qlogs will only contain a single trace in this array. These can later be combined into a single qlog file by taking the "traces" entry/entries for each qlog file individually and copying them to the "traces" array of a new, aggregated qlog file. This is typically done in a post-processing step.

The "traces" array can thus contain both normal traces (for the definition of the Trace type, see Section 3.3), but also "error" entries. These indicate that we tried to find/convert a file for inclusion in the aggregated qlog, but there was an error during the process. Rather than silently dropping the erroneous file, we can opt to explicitly include it in the qlog file as an entry in the "traces" array, as shown in Figure 6.

Definition:

```
TraceError = {  
  error_description: text  
  ; the original URI at which we attempted to find the file  
  ? uri: text  
  ? vantage_point: VantagePoint  
}
```

Figure 6: TraceError definition

JSON serialization example:

```
{  
  "error_description": "File could not be found",  
  "uri": "/srv/traces/today/latest.qlog",  
  "vantage_point": { type: "server" }  
}
```

Figure 7: TraceError example

Note that another way to combine events of different traces in a single qlog file is through the use of the "group_id" field, discussed in Section 3.4.6.

3.3. Individual Trace containers

The exact conceptual definition of a Trace can be fluid. For example, a trace could contain all events for a single connection, for a single endpoint, for a single measurement interval, for a single protocol, etc. As such, a Trace container contains some metadata in addition to the logged events, see Figure 8.

In the normal use case however, a trace is a log of a single data flow collected at a single location or vantage point. For example, for QUIC, a single trace only contains events for a single logical QUIC connection for either the client or the server.

The semantics and context of the trace can mainly be deduced from the entries in the "common_fields" list and "vantage_point" field.

Definition:

```
Trace = {  
  ? title: text  
  ? description: text  
  ? configuration: Configuration  
  ? common_fields: CommonFields  
  ? vantage_point: VantagePoint  
  events: [* Event]  
}
```

Figure 8: Trace definition

JSON serialization example:

```
{  
  "title": "Name of this particular trace (short)",  
  "description": "Description for this trace (long)",  
  "configuration": {  
    "time_offset": 150  
  },  
  "common_fields": {  
    "ODCID": "abcde1234",  
    "time_format": "absolute"  
  },  
  "vantage_point": {  
    "name": "backend-67",  
    "type": "server"  
  },  
  "events": [...]  
}
```

Figure 9: Trace example

3.3.1. Configuration

We take into account that a qlog file is usually not used in isolation, but by means of various tools. Especially when aggregating various traces together or preparing traces for a demonstration, one might wish to persist certain tool-based settings inside the qlog file itself. For this, the configuration field is used.

The configuration field can be viewed as a generic metadata field that tools can fill with their own fields, based on per-tool logic. It is best practice for tools to prefix each added field with their tool name to prevent collisions across tools. This document only defines two optional, standard, tool-independent configuration settings: "time_offset" and "original_uris".

Definition:

```
Configuration = {  
    ; time_offset is in milliseconds  
    time_offset: float64  
    original_uris:[* text]  
    * text => any  
}
```

Figure 10: Configuration definition

JSON serialization example:

```
{  
  "time_offset": 150,  
  "original_uris": [  
    "https://example.org/trace1.qlog",  
    "https://example.org/trace2.qlog"  
  ]  
}
```

Figure 11: Configuration example

3.3.1.1. time_offset

The `time_offset` field indicates by how many milliseconds the starting time of the current trace should be offset. This is useful when comparing logs taken from various systems, where clocks might not be perfectly synchronous. Users could use manual tools or automated logic to align traces in time and the found optimal offsets can be stored in this field for future usage. The default value is 0.

3.3.1.2. original_uris

The `original_uris` field is used when merging multiple individual qlog files or other source files (e.g., when converting .pcaps to qlog). It allows to keep better track where certain data came from. It is a simple array of strings. It is an array instead of a single string, since a single qlog trace can be made up out of an aggregation of multiple component qlog traces as well. The default value is an empty array.

3.3.1.3. custom fields

Tools can add optional custom metadata to the "configuration" field to store state and make it easier to share specific data viewpoints and view configurations.

Two examples from the qvis toolset (<https://qvis.edm.uhasselt.be>) are shown in Figure 12.

```
{
  "configuration" : {
    "qvis" : {
      "congestion_graph": {
        "startX": 1000,
        "endX": 2000,
        "focusOnEventIndex": 124
      }
      "sequence_diagram" : {
        "focusOnEventIndex": 555
      }
    }
  }
}
```

Figure 12: Custom configuration fields example

3.3.2. vantage_point

The `vantage_point` field describes the vantage point from which the trace originates, see Figure 13. Each trace can have only a single `vantage_point` and thus all events in a trace MUST BE from the perspective of this `vantage_point`. To include events from multiple `vantage_points`, implementers can for example include multiple traces, split by `vantage_point`, in a single qlog file.

Definitions:

```
VantagePoint = {
  ? name: text
  type: VantagePointType
  ? flow: VantagePointType
}

; client = endpoint which initiates the connection
; server = endpoint which accepts the connection
; network = observer in between client and server
VantagePointType = "client" / "server" / "network" / "unknown"
```

Figure 13: VantagePoint definition

JSON serialization examples:

```
{
  "name": "aioquic client",
  "type": "client",
}

{
  "name": "wireshark trace",
  "type": "network",
  "flow": "client"
}
```

Figure 14: VantagePoint example

The flow field is only required if the type is "network" (for example, the trace is generated from a packet capture). It is used to disambiguate events like "packet sent" and "packet received". This is indicated explicitly because for multiple reasons (e.g., privacy) data from which the flow direction can be otherwise inferred (e.g., IP addresses) might not be present in the logs.

Meaning of the different values for the flow field: * "client" indicates that this vantage point follows client data flow semantics (a "packet sent" event goes in the direction of the server). * "server" indicates that this vantage point follow server data flow semantics (a "packet sent" event goes in the direction of the client). * "unknown" indicates that the flow's direction is unknown.

Depending on the context, tools confronted with "unknown" values in the vantage_point can either try to heuristically infer the semantics from protocol-level domain knowledge (e.g., in QUIC, the client always sends the first packet) or give the user the option to switch between client and server perspectives manually.

3.4. Field name semantics

Inside of the "events" field of a qlog trace is a list of events logged by the endpoint. Each event is specified as a generic object with a number of member fields and their associated data. Depending on the protocol and use case, the exact member field names and their formats can differ across implementations. This section lists the main, pre-defined and reserved field names with specific semantics and expected corresponding value formats.

Each qlog event at minimum requires the "time" (Section 3.4.1), "name" (Section 3.4.2) and "data" (Section 3.4.3) fields. Other typical fields are "time_format" (Section 3.4.1), "protocol_type" (Section 3.4.4), "trigger" (Section 3.4.5), and "group_id" (Section 3.4.6). As especially these later fields typically have identical values across individual event instances, they are normally logged separately in the "common_fields" (Section 3.4.7).

The specific values for each of these fields and their semantics are defined in separate documents, specific per protocol or use case. For example: event definitions for QUIC, HTTP/3 and QPACK can be found in [QLOG-QUIC] and [QLOG-H3].

Other fields are explicitly allowed by the qlog approach, and tools SHOULD allow for the presence of unknown event fields, but their semantics depend on the context of the log usage (e.g., for QUIC, the ODCID field is used), see [QLOG-QUIC].

An example of a qlog event with its component fields is shown in Figure 15.

Definition:

```
Event = {  
  time: float64  
  name: text  
  data: $ProtocolEventBody  
  
  ? time_format: TimeFormat  
  
  ? protocol_type: ProtocolType  
  ? group_id: GroupID  
  
  ; events can contain any amount of custom fields  
  * text => any  
}
```

Figure 15: Event definition

JSON serialization:

```
{
  time: 1553986553572,

  name: "transport:packet_sent",
  data: { ... }

  protocol_type: ["QUIC","HTTP3"],
  group_id: "127ecc830d98f9d54a42c4f0842aa87e181a",

  time_format: "absolute",

  ODCID: "127ecc830d98f9d54a42c4f0842aa87e181a",
}
```

Figure 16: Event example

3.4.1. Timestamps

The "time" field indicates the timestamp at which the event occurred. Its value is typically the Unix timestamp since the 1970 epoch (number of milliseconds since midnight UTC, January 1, 1970, ignoring leap seconds). However, qlog supports two more succinct timestamps formats to allow reducing file size. The employed format is indicated in the "time_format" field, which allows one of three values: "absolute", "delta" or "relative".

Definition:

TimeFormat = "absolute" / "delta" / "relative"

Figure 17: TimeFormat definition

- * Absolute: Include the full absolute timestamp with each event. This approach uses the largest amount of characters. This is also the default value of the "time_format" field.
- * Delta: Delta-encode each time value on the previously logged value. The first event in a trace typically logs the full absolute timestamp. This approach uses the least amount of characters.
- * Relative: Specify a full "reference_time" timestamp (typically this is done up-front in "common_fields", see Section 3.4.7) and include only relatively-encoded values based on this reference_time with each event. The "reference_time" value is typically the first absolute timestamp. This approach uses a medium amount of characters.

The first option is good for stateless loggers, the second and third for stateful loggers. The third option is generally preferred, since it produces smaller files while being easier to reason about. An example for each option can be seen in Figure 18.

The absolute approach will use:
1500, 1505, 1522, 1588

The delta approach will use:
1500, 5, 17, 66

The relative approach will:
- set the reference_time to 1500 in "common_fields"
- use: 0, 5, 22, 88

Figure 18: Three different approaches for logging timestamps

One of these options is typically chosen for the entire trace (put differently: each event has the same value for the "time_format" field). Each event MUST include a timestamp in the "time" field.

Events in each individual trace SHOULD be logged in strictly ascending timestamp order (though not necessarily absolute value, for the "delta" format). Tools CAN sort all events on the timestamp before processing them, though are not required to (as this could impose a significant processing overhead). This can be a problem especially for multi-threaded and/or streaming loggers, who could consider using a separate postprocessor to order qlog events in time if a tool do not provide this feature.

Timestamps do not have to use the UNIX epoch timestamp as their reference. For example for privacy considerations, any initial reference timestamps (for example "endpoint uptime in ms" or "time since connection start in ms") can be chosen. Tools SHOULD NOT assume the ability to derive the absolute Unix timestamp from qlog traces, nor allow on them to relatively order events across two or more separate traces (in this case, clock drift should also be taken into account).

3.4.2. Category and Event Type

Events differ mainly in the type of metadata associated with them. To help identify a given event and how to interpret its metadata in the "data" field (see Section 3.4.3), each event has an associated "name" field. This can be considered as a concatenation of two other fields, namely event "category" and event "type".

Category allows a higher-level grouping of events per specific event type. For example for QUIC and HTTP/3, the different categories could be "transport", "http", "qpack", and "recovery". Within these categories, the event Type provides additional granularity. For example for QUIC and HTTP/3, within the "transport" Category, there would be "packet_sent" and "packet_received" events.

Logging category and type separately conceptually allows for fast and high-level filtering based on category and the re-use of event types across categories. However, it also considerably inflates the log size and this flexibility is not used extensively in practice at the time of writing.

As such, the default approach in qlog is to concatenate both field values using the ":" character in the "name" field, as can be seen in Figure 19. As such, qlog category and type names MUST NOT include this character.

JSON serialization using separate fields:

```
{
  "category": "transport",
  "type": "packet_sent"
}
```

JSON serialization using ":" concatenated field:

```
{
  "name": "transport:packet_sent"
}
```

Figure 19: Ways of logging category, type and name of an event.

Certain serializations CAN emit category and type as separate fields, and qlog tools SHOULD be able to deal with both the concatenated "name" field, and the separate "category" and "type" fields. Text-based serializations however are encouraged to employ the concatenated "name" field for efficiency.

3.4.3. Data

The data field is a generic object. It contains the per-event metadata and its form and semantics are defined per specific sort of event. For example, data field value definitions for QUIC and HTTP/3 can be found in [QLOG-QUIC] and [QLOG-H3].

This field is defined here as a CDDL extension point (a "socket" or "plug") named \$ProtocolEventBody. Other documents MUST properly extend this extension point when defining new data field content options to enable automated validation of aggregated qlog schemas.

The only common field defined for the data field is the trigger field, which is discussed in Section 3.4.5.

Definition:

```
; The ProtocolEventBody is any key-value map (e.g., JSON object)
; only the optional trigger field is defined in this document
$ProtocolEventBody /= {
    ? trigger: text
    * text => any
}
; event documents are intended to extend this socket by using:
; NewProtocolEvents = EventType1 / EventType2 / ... / EventTypeN
; $ProtocolEventBody /= NewProtocolEvents
```

Figure 20: ProtocolEventBody definition

One purely illustrative example for a QUIC "packet_sent" event is shown in Figure 21:

```
TransportPacketSent = {
    ? packet_size: uint16
    header: PacketHeader
    ? frames:[* QuicFrame]
    ? trigger: "pto_probe" / "retransmit_timeout" / "bandwidth_probe"
}
```

could be serialized as

```
{
  packet_size: 1280,
  header: {
    packet_type: "1RTT",
    packet_number: 123
  },
  frames: [
    {
      frame_type: "stream",
      length: 1000,
      offset: 456
    },
    {
      frame_type: "padding"
    }
  ]
}
```

Figure 21: Example of the 'data' field for a QUIC packet_sent event

3.4.4. protocol_type

The "protocol_type" array field indicates to which protocols (or protocol "stacks") this event belongs. This allows a single qlog file to aggregate traces of different protocols (e.g., a web server offering both TCP+HTTP/2 and QUIC+HTTP/3 connections).

Definition:

ProtocolType = [+ text]

Figure 22: ProtocolType definition

For example, QUIC and HTTP/3 events have the "QUIC" and "HTTP3" protocol_type entry values, see [QLOG-QUIC] and [QLOG-H3].

Typically however, all events in a single trace are of the same few protocols, and this array field is logged once in "common_fields", see Section 3.4.7.

3.4.5. Triggers

Sometimes, additional information is needed in the case where a single event can be caused by a variety of other events. In the normal case, the context of the surrounding log messages gives a hint as to which of these other events was the cause. However, in highly-parallel and optimized implementations, corresponding log messages might separated in time. Another option is to explicitly indicate these "triggers" in a high-level way per-event to get more fine-grained information without much additional overhead.

In qlog, the optional "trigger" field contains a string value describing the reason (if any) for this event instance occurring, see Section 3.4.3. While this "trigger" field could be a property of the qlog Event itself, it is instead a property of the "data" field instead. This choice was made because many event types do not include a trigger value, and having the field at the Event-level would cause overhead in some serializations. Additional information on the trigger can be added in the form of additional member fields of the "data" field value, yet this is highly implementation-specific, as are the trigger field's string values.

One purely illustrative example of some potential triggers for QUIC's "packet_dropped" event is shown in Figure 23:

```
TransportPacketDropped = {  
  ? packet_type: PacketType  
  ? raw_length: uint16  
  
  ? trigger: "key_unavailable" / "unknown_connection_id" /  
            "decrypt_error" / "unsupported_version"  
}
```

Figure 23: Trigger example

3.4.6. group_id

As discussed in Section 3.3, a single qlog file can contain several traces taken from different vantage points. However, a single trace from one endpoint can also contain events from a variety of sources. For example, a server implementation might choose to log events for all incoming connections in a single large (streamed) qlog file. As such, we need a method for splitting up events belonging to separate logical entities.

The simplest way to perform this splitting is by associating a "group identifier" to each event that indicates to which conceptual "group" each event belongs. A post-processing step can then extract events per group. However, this group identifier can be highly protocol and context-specific. In the example above, we might use QUIC's "Original Destination Connection ID" to uniquely identify a connection. As such, they might add a "ODCID" field to each event. However, a middlebox logging IP or TCP traffic might rather use four-tuples to identify connections, and add a "four_tuple" field.

As such, to provide consistency and ease of tooling in cross-protocol and cross-context setups, qlog instead defines the common "group_id" field, which contains a string value. Implementations are free to use their preferred string serialization for this field, so long as it contains a unique value per logical group. Some examples can be seen in Figure 25.

Definition:

```
GroupID = text
```

Figure 24: GroupID definition

JSON serialization example for events grouped by four tuples and QUIC connection IDs:

```
events: [  
  {  
    time: 1553986553579,  
    protocol_type: ["TCP", "TLS", "HTTP2"],  
    group_id: "ip1=2001:67c:1232:144:9498:6df6:f450:110b,  
              ip2=2001:67c:2b0:1c1::198,port1=59105,port2=80",  
    name: "transport:packet_received",  
    data: { ... },  
  },  
  {  
    time: 1553986553581,  
    protocol_type: ["QUIC", "HTTP3"],  
    group_id: "127ecc830d98f9d54a42c4f0842aa87e181a",  
    name: "transport:packet_sent",  
    data: { ... },  
  }  
]
```

Figure 25: GroupID example

Note that in some contexts (for example a Multipath transport protocol) it might make sense to add additional contextual per-event fields (for example "path_id"), rather than use the group_id field for that purpose.

Note also that, typically, a single trace only contains events belonging to a single logical group (for example, an individual QUIC connection). As such, instead of logging the "group_id" field with an identical value for each event instance, this field is typically logged once in "common_fields", see Section 3.4.7.

3.4.7. common_fields

As discussed in the previous sections, information for a typical qlog event varies in three main fields: "time", "name" and associated data. Additionally, there are also several more advanced fields that allow mixing events from different protocols and contexts inside of the same trace (for example "protocol_type" and "group_id"). In most "normal" use cases however, the values of these advanced fields are consistent for each event instance (for example, a single trace contains events for a single QUIC connection).

To reduce file size and making logging easier, qlog uses the "common_fields" list to indicate those fields and their values that are shared by all events in this component trace. This prevents these fields from being logged for each individual event. An example of this is shown in Figure 26.

JSON serialization with repeated field values
per-event instance:

```
{
  events: [{
    group_id: "127ecc830d98f9d54a42c4f0842aa87e181a",
    protocol_type: ["QUIC", "HTTP3"],
    time_format: "relative",
    reference_time: 1553986553572,

    time: 2,
    name: "transport:packet_received",
    data: { ... }
  }, {
    group_id: "127ecc830d98f9d54a42c4f0842aa87e181a",
    protocol_type: ["QUIC", "HTTP3"],
    time_format: "relative",
    reference_time: 1553986553572,

    time: 7,
    name: "http:frame_parsed",
    data: { ... }
  }
]
```

JSON serialization with repeated field values instead
extracted to common_fields:

```
{
  common_fields: {
    group_id: "127ecc830d98f9d54a42c4f0842aa87e181a",
    protocol_type: ["QUIC", "HTTP3"],
    time_format: "relative",
    reference_time: 1553986553572
  },
  events: [
    {
      time: 2,
      name: "transport:packet_received",
      data: { ... }
    }, {
      7,
      name: "http:frame_parsed",
      data: { ... }
    }
  ]
}
```

Figure 26: CommonFields example

The "common_fields" field is a generic dictionary of key-value pairs, where the key is always a string and the value can be of any type, but is typically also a string or number. As such, unknown entries in this dictionary MUST be disregarded by the user and tools (i.e., the presence of an unknown field is explicitly NOT an error).

The list of default qlog fields that are typically logged in common_fields (as opposed to as individual fields per event instance) are shown in the listing below:

Definition:

```
CommonFields = {  
    ? time_format: TimeFormat  
    ? reference_time: float64  
  
    ? protocol_type: ProtocolType  
    ? group_id: GroupID  
  
    * text => any  
}
```

Figure 27: CommonFields definition

Tools MUST be able to deal with these fields being defined either on each event individually or combined in common_fields. Note that if at least one event in a trace has a different value for a given field, this field MUST NOT be added to common_fields but instead defined on each event individually. Good example of such fields are "time" and "data", who are divergent by nature.

4. Guidelines for event definition documents

This document only defines the main schema for the qlog format. This is intended to be used together with specific, per-protocol event definitions that specify the name (category + type) and data needed for each individual event. This is with the intent to allow the qlog main schema to be easily re-used for several protocols. Examples include the QUIC event definitions [QLOG-QUIC] and HTTP/3 and QPACK event definitions [QLOG-H3].

This section defines some basic annotations and concepts the creators of event definition documents SHOULD follow to ensure a measure of consistency, making it easier for qlog implementers to extrapolate from one protocol to another.

4.1. Event design guidelines

TODO: pending QUIC working group discussion. This text reflects the initial (qlog draft 01 and 02) setup.

There are several ways of defining qlog events. In practice, we have seen two main types used so far: a) those that map directly to concepts seen in the protocols (e.g., `packet_sent`) and b) those that act as aggregating events that combine data from several possible protocol behaviours or code paths into one (e.g., `parameters_set`). The latter are typically used as a means to reduce the amount of unique event definitions, as reflecting each possible protocol event as a separate qlog entity would cause an explosion of event types.

Additionally, logging duplicate data is typically prevented as much as possible. For example, packet header values that remain consistent across many packets are split into separate events (for example `spin_bit_updated` or `connection_id_updated` for QUIC).

Finally, we have typically refrained from adding additional state change events if those state changes can be directly inferred from data on the wire (for example flow control limit changes) if the implementation is bug-free and spec-compliant. Exceptions have been made for common events that benefit from being easily identifiable or individually logged (for example `packets_acked`).

4.2. Event importance indicators

Depending on how events are designed, it may be that several events allow the logging of similar or overlapping data. For example the separate QUIC `connection_started` event overlaps with the more generic `connection_state_updated`. In these cases, it is not always clear which event should be logged or used, and which event should take precedence if e.g., both are present and provide conflicting information.

To aid in this decision making, we recommend that each event SHOULD have an "importance indicator" with one of three values, in decreasing order of importance and expected usage:

- * Core
- * Base
- * Extra

The "Core" events are the events that SHOULD be present in all qlog files for a given protocol. These are typically tied to basic packet and frame parsing and creation, as well as listing basic internal metrics. Tool implementers SHOULD expect and add support for these events, though SHOULD NOT expect all Core events to be present in each qlog trace.

The "Base" events add additional debugging options and CAN be present in qlog files. Most of these can be implicitly inferred from data in Core events (if those contain all their properties), but for many it is better to log the events explicitly as well, making it clearer how the implementation behaves. These events are for example tied to passing data around in buffers, to how internal state machines change and help show when decisions are actually made based on received data. Tool implementers SHOULD at least add support for showing the contents of these events, if they do not handle them explicitly.

The "Extra" events are considered mostly useful for low-level debugging of the implementation, rather than the protocol. They allow more fine-grained tracking of internal behaviour. As such, they CAN be present in qlog files and tool implementers CAN add support for these, but they are not required to.

Note that in some cases, implementers might not want to log for example data content details in the "Core" events due to performance or privacy considerations. In this case, they SHOULD use (a subset of) relevant "Base" events instead to ensure usability of the qlog output. As an example, implementations that do not log QUIC packet_received events and thus also not which (if any) ACK frames the packet contains, SHOULD log packets_acked events instead.

Finally, for event types whose data (partially) overlap with other event types' definitions, where necessary the event definition document should include explicit guidance on which to use in specific situations.

4.3. Custom fields

Event definition documents are free to define new category and event types, top-level fields (e.g., a per-event field indicating its privacy properties or path_id in multipath protocols), as well as values for the "trigger" property within the "data" field, or other member fields of the "data" field, as they see fit.

They however SHOULD NOT expect non-specialized tools to recognize or visualize this custom data. However, tools SHOULD make an effort to visualize even unknown data if possible in the specific tool's context. If they do not, they MUST ignore these unknown fields.

5. Generic events and data classes

There are some event types and data classes that are common across protocols, applications and use cases that benefit from being defined in a single location. This section specifies such common definitions.

5.1. Raw packet and frame information

While qlog is a more high-level logging format, it also allows the inclusion of most raw wire image information, such as byte lengths and even raw byte values. This can be useful when for example investigating or tuning packetization behaviour or determining encoding/framing overheads. However, these fields are not always necessary and can take up considerable space if logged for each packet or frame. They can also have a considerable privacy and security impact. As such, they are grouped in a separate optional field called "raw" of type RawInfo (where applicable).

Definition:

```
RawInfo = {  
    ; the full byte length of the entity (e.g., packet or frame),  
    ; including headers and trailers  
    ? length: uint64  
  
    ; the byte length of the entity's payload,  
    ; without headers or trailers  
    ? payload_length: uint64  
  
    ; the contents of the full entity,  
    ; including headers and trailers  
    ? data: hexstring  
}
```

Figure 28: RawInfo definition

Note: The RawInfo:data field can be truncated for privacy or security purposes (for example excluding payload data), see Section 6.1.2. In this case, the length properties should still indicate the non-truncated lengths.

Note: We do not specify explicit header_length or trailer_length

fields. In most protocols, header_length can be calculated by subtracting the payload_length from the length (e.g., if trailer_length is always 0). In protocols with trailers (e.g., QUIC's AEAD tag), event definitions documents SHOULD define other ways of logging the trailer_length to make the header_length calculation possible.

The exact definitions entities, headers, trailers and payloads depend on the protocol used. If this is non-trivial, event definitions documents SHOULD include a clear explanation of how entities are mapped into the RawInfo structure.

Note: Relatedly, many modern protocols use Variable-Length Integer Encoded (VLIE) values in their headers, which are of a dynamic length. Because of this, we cannot deterministically reconstruct the header encoding/length from non-RawInfo qlog data, as implementations might not necessarily employ the most efficient VLIE scheme for all values. As such, to make exact size-analysis possible, implementers should use explicit lengths in RawInfo rather than reconstructing them from other qlog data. Similarly, tool developers should only utilize RawInfo (and related information) in such tools to prevent errors.

5.2. Generic events

In typical logging setups, users utilize a discrete number of well-defined logging categories, levels or severities to log freeform (string) data. This generic events category replicates this approach to allow implementations to fully replace their existing text-based logging by qlog. This is done by providing events to log generic strings for the typical well-known logging levels (error, warning, info, debug, verbose).

For the events defined below, the "category" is "generic" and their "type" is the name of the heading in lowercase (e.g., the "name" of the error event is "generic:error").

5.2.1. error

Importance: Core

Used to log details of an internal error that might not get reflected on the wire.

Definition:

```
GenericError = {  
  ? code: uint64  
  ? message: text  
}
```

Figure 29: GenericError definition

5.2.2. warning

Importance: Base

Used to log details of an internal warning that might not get reflected on the wire.

Definition:

```
GenericWarning = {  
  ? code: uint64  
  ? message: text  
}
```

Figure 30: GenericWarning definition

5.2.3. info

Importance: Extra

Used mainly for implementations that want to use qlog as their one and only logging format but still want to support unstructured string messages.

Definition:

```
GenericInfo = {  
  message: text  
}
```

Figure 31: GenericInfo definition

5.2.4. debug

Importance: Extra

Used mainly for implementations that want to use qlog as their one and only logging format but still want to support unstructured string messages.

Definition:

```
GenericDebug = {  
  message: text  
}
```

Figure 32: GenericDebug definition

5.2.5. verbose

Importance: Extra

Used mainly for implementations that want to use qlog as their one and only logging format but still want to support unstructured string messages.

Definition:

```
GenericVerbose = {  
  message: text  
}
```

Figure 33: GenericVerbose definition

5.3. Simulation events

When evaluating a protocol implementation, one typically sets up a series of interoperability or benchmarking tests, in which the test situations can change over time. For example, the network bandwidth or latency can vary during the test, or the network can be fully disable for a short time. In these setups, it is useful to know when exactly these conditions are triggered, to allow for proper correlation with other events.

For the events defined below, the "category" is "simulation" and their "type" is the name of the heading in lowercase (e.g., the "name" of the scenario event is "simulation:scenario").

5.3.1. scenario

Importance: Extra

Used to specify which specific scenario is being tested at this particular instance. This could also be reflected in the top-level qlog's summary or configuration fields, but having a separate event allows easier aggregation of several simulations into one trace (e.g., split by group_id).

Definition:


```
SimulationScenario = {  
  ? name: text  
  ? details: { * text => any }  
}
```

Figure 34: SimulationScenario definition

5.3.2. marker

Importance: Extra

Used to indicate when specific emulation conditions are triggered at set times (e.g., at 3 seconds in 2% packet loss is introduced, at 10s a NAT rebind is triggered).

Definition:

```
SimulationMarker = {  
  ? type: text  
  ? message: text  
}
```

Figure 35: SimulationMarker definition

6. Serializing qlog

This document and other related qlog schema definitions are intentionally serialization-format agnostic. This means that implementers themselves can choose how to represent and serialize qlog data practically on disk or on the wire. Some examples of possible formats are JSON, CBOR, CSV, protocol buffers, flatbuffers, etc.

All these formats make certain tradeoffs between flexibility and efficiency, with textual formats like JSON typically being more flexible but also less efficient than binary formats like protocol buffers. The format choice will depend on the practical use case of the qlog user. For example, for use in day to day debugging, a plaintext readable (yet relatively large) format like JSON is probably preferred. However, for use in production, a more optimized yet restricted format can be better. In this latter case, it will be more difficult to achieve interoperability between qlog implementations of various protocol stacks, as some custom or tweaked events from one might not be compatible with the format of the other. This will also reflect in tooling: not all tools will support all formats.

This being said, the authors prefer JSON as the basis for storing qlog, as it retains full flexibility and maximum interoperability. Storage overhead can be managed well in practice by employing compression. For this reason, this document details how to practically transform qlog schema definitions to [JSON], its subset [I-JSON], and its streamable derivative [JSON-Text-Sequences]s. We discuss concrete options to bring down JSON size and processing overheads in Section 6.3.

As depending on the employed format different deserializers/parsers should be used, the "qlog_format" field is used to indicate the chosen serialization approach. This field is always a string, but can be made hierarchical by the use of the "." separator between entries. For example, a value of "JSON.optimizationA" can indicate that a default JSON format is being used, but that a certain optimization of type A was applied to the file as well (see also Section 6.3).

6.1. qlog to JSON mapping

When mapping qlog to normal JSON, the "qlog_format" field MUST have the value "JSON". This is also the default qlog serialization and default value of this field.

When using normal JSON serialization, the file extension/suffix SHOULD be ".qlog" and the Media Type (if any) SHOULD be "application/qlog+json" per [RFC6839].

JSON files by definition ([RFC8259]) MUST utilize the UTF-8 encoding, both for the file itself and the string values.

While not specifically required by the JSON specification, all qlog field names in a JSON serialization MUST be lowercase.

In order to serialize CDDL-based qlog event and data structure definitions to JSON, the official CDDL-to-JSON mapping defined in Appendix E of [CDDL] SHOULD be employed.

6.1.1. I-JSON

For some use cases, it should be taken into account that not all popular JSON parsers support the full JSON format. Especially for parsers integrated with the JavaScript programming language (e.g., Web browsers, NodeJS), users are recommended to stick to a JSON subset dubbed [I-JSON] (or Internet-JSON).

One of the key limitations of JavaScript and thus I-JSON is that it cannot represent full 64-bit integers in standard operating mode (i.e., without using BigInt extensions), instead being limited to the range of $[-(2^{53})+1, (2^{53})-1]$. In these circumstances, Appendix E of [CDDL] recommends defining new CDDL types for int64 and uint64 that limit their values to this range.

While this can be sensible and workable for most use cases, some protocols targeting qlog serialization (e.g., QUIC, HTTP/3), might require full uint64 variables in some (rare) circumstances. In these situations, it should be allowed to also use the string-based representation of uint64 values alongside the numerical representation. Concretely, the following definition of uint64 should override the original and (web-based) tools should take into account that a uint64 field can be either a number or string.

```
uint64 = text / uint .size 8
```

Figure 36: Custom uint64 definition for I-JSON

6.1.2. Truncated values

For some use cases (e.g., limiting file size, privacy), it can be necessary not to log a full raw blob (using the hexstring type) but instead a truncated value (for example, only the first 100 bytes of an HTTP response body to be able to discern which file it actually contained). In these cases, the original byte-size length cannot be obtained from the serialized value directly.

As such, all qlog schema definitions SHOULD include a separate, length-indicating field for all fields of type hexstring they specify, see for example Section 5.1. This not only ensures the original length can always be retrieved, but also allows the omission of any raw value bytes of the field completely (e.g., out of privacy or security considerations).

To reduce overhead however and in the case the full raw value is logged, the extra length-indicating field can be left out. As such, tools MUST be able to deal with this situation and derive the length of the field from the raw value if no separate length-indicating field is present. The main possible permutations are shown by example in Figure 37.

```
// both the full raw value and its length are present
// (length is redundant)
{
  "raw_length": 5,
  "raw": "051428abff"
}

// only the raw value is present, indicating it
// represents the fields full value the byte
// length is obtained by calculating raw.length / 2
{
  "raw": "051428abff"
}

// only the length field is present, meaning the
// value was omitted
{
  "raw_length": 5,
}

// both fields are present and the lengths do not match:
// the value was truncated to the first three bytes.
{
  "raw_length": 5,
  "raw": "051428"
}
```

Figure 37: Example for serializing truncated hexstrings

6.2. qlog to JSON Text Sequences mapping

One of the downsides of using pure JSON is that it is inherently a non-streamable format. Put differently, it is not possible to simply append new qlog events to a log file without "closing" this file at the end by appending "}}}}". Without these closing tags, most JSON parsers will be unable to parse the file entirely. As most platforms do not provide a standard streaming JSON parser (which would be able to deal with this problem), this document also provides a qlog mapping to a streamable JSON format called JSON Text Sequences (JSON-SEQ) ([RFC7464]).

When mapping qlog to JSON-SEQ, the "qlog_format" field MUST have the value "JSON-SEQ".

When using JSON-SEQ serialization, the file extension/suffix SHOULD be ".sqlog" (for "streaming" qlog) and the Media Type (if any) SHOULD be "application/qlog+json-seq" per [RFC8091].

JSON Text Sequences are very similar to JSON, except that JSON objects are serialized as individual records, each prefixed by an ASCII Record Separator (<RS>, 0x1E), and each ending with an ASCII Line Feed character (\n, 0x0A). Note that each record can also contain any amount of newlines in its body, as long as it ends with a newline character before the next <RS> character.

Each qlog event is serialized and interpreted as an individual JSON Text Sequence record, and can simply be appended as a new object at the back of an event stream or log file. Put differently, unlike default JSON, it does not require a file to be wrapped as a full object with "{ ... }" or "[...]".

For this to work, some qlog definitions have to be adjusted however. Mainly, events are no longer part of the "events" array in the Trace object, but are instead logged separately from the qlog "header", as indicated by the TraceSeq object in Figure 38. Additionally, qlog's JSON-SEQ mapping does not allow logging multiple individual traces in a single qlog file. As such, the QlogFile:traces field is replaced by the singular QlogFileSeq:trace field, see Figure 39. An example can be seen in Figure 40. Note that the "group_id" field can still be used on a per-event basis to include events from conceptually different sources in a single JSON-SEQ qlog file.

Definition:

```
TraceSeq = {
  ? title: text
  ? description: text
  ? configuration: Configuration
  ? common_fields: CommonFields
  ? vantage_point: VantagePoint
}
```

Figure 38: TraceSeq definition

Definition:

```
QlogFileSeq = {
  qlog_format: "JSON-SEQ"

  qlog_version: text
  ? title: text
  ? description: text
  ? summary: Summary
  trace: TraceSeq
}
```

Figure 39: QlogFileSeq definition

JSON-SEQ serialization examples:

```
// list of qlog events, serialized in accordance with RFC 7464,
// starting with a Record Separator character and ending with a
// newline.
// For display purposes, Record Separators are rendered as <RS>

<RS>{
  "qlog_version": "0.3",
  "qlog_format": "JSON-SEQ",
  "title": "Name of JSON Text Sequence qlog file (short)",
  "description": "Description for this trace file (long)",
  "summary": {
    ...
  },
  "trace": {
    "common_fields": {
      "protocol_type": ["QUIC", "HTTP3"],
      "group_id": "127ecc830d98f9d54a42c4f0842aa87e181a",
      "time_format": "relative",
      "reference_time": 1553986553572
    },
    "vantage_point": {
      "name": "backend-67",
      "type": "server"
    }
  }
}
<RS>{"time": 2, "name": "transport:parameters_set", "data": { ... } }
<RS>{"time": 7, "name": "transport:packet_sent", "data": { ... } }
...
```

Figure 40: Top-level element

Note: while not specifically required by the JSON-SEQ specification, all qlog field names in a JSON-SEQ serialization MUST be lowercase.

In order to serialize all other CDDL-based qlog event and data structure definitions to JSON-SEQ, the official CDDL-to-JSON mapping defined in Appendix E of [CDDL] SHOULD still be employed.

6.2.1. Supporting JSON Text Sequences in tooling

Note that JSON Text Sequences are not supported in most default programming environments (unlike normal JSON). However, several custom JSON-SEQ parsing libraries exist in most programming languages that can be used and the format is easy enough to parse with existing implementations (i.e., by splitting the file into its component records and feeding them to a normal JSON parser individually, as each record by itself is a valid JSON object).

6.3. Other optimized formatting options

Both the JSON and JSON-SEQ formatting options described above are serviceable in general small to medium scale (debugging) setups. However, these approaches tend to be relatively verbose, leading to larger file sizes. Additionally, generalized JSON(-SEQ) (de)serialization performance is typically (slightly) lower than that of more optimized and predictable formats. Both aspects make these formats more challenging (though still practical (<https://qlog.edm.uhasselt.be/anrw/>)) to use in large scale setups.

During the development of qlog, we compared a multitude of alternative formatting and optimization options. The results of this study are summarized on the qlog github repository (<https://github.com/quiclog/internet-drafts/issues/30#issuecomment-617675097>). The rest of this section discusses some of these approaches implementations could choose and the expected gains and tradeoffs inherent therein. Tools SHOULD support mainly the compression options listed in Section 6.3.2, as they provide the largest wins for the least cost overall.

Over time, specific qlog formats and encodings can be created that more formally define and combine some of the discussed optimizations or add new ones. We choose to define these schemes in separate documents to keep the main qlog definition clean and generalizable, as not all contexts require the same performance or flexibility as others and qlog is intended to be a broadly usable and extensible format (for example more flexibility is needed in earlier stages of protocol development, while more performance is typically needed in later stages). This is also the main reason why the general qlog format is the less optimized JSON instead of a more performant option.

To be able to easily distinguish between these options in qlog compatible tooling (without the need to have the user provide out-of-band information or to (heuristically) parse and process files in a multitude of ways, see also Section 8), we recommend using explicit file extensions to indicate specific formats. As there are no

standards in place for this type of extension to format mapping, we employ a commonly used scheme here. Our approach is to list the applied optimizations in the extension in ascending order of application (e.g., if a qlog file is first optimized with technique A and then compressed with technique B, the resulting file would have the extension ".(s)qlog.A.B"). This allows tooling to start at the back of the extension to "undo" applied optimizations to finally arrive at the expected qlog representation.

6.3.1. Data structure optimizations

The first general category of optimizations is to alter the representation of data within an JSON(-SEQ) qlog file to reduce file size.

The first option is to employ a scheme similar to the CSV (comma separated value [RFC4180]) format, which utilizes the concept of column "headers" to prevent repeating field names for each datapoint instance. Concretely for JSON qlog, several field names are repeated with each event (i.e., time, name, data). These names could be extracted into a separate list, after which qlog events could be serialized as an array of values, as opposed to a full object. This approach was a key part of the original qlog format (prior to draft-02) using the "event_fields" field. However, tests showed that this optimization only provided a mean file size reduction of 5% (100MB to 95MB) while significantly increasing the implementation complexity, and this approach was abandoned in favor of the default JSON setup. Implementations using this format should not employ a separate file extension (as it still uses JSON), but rather employ a new value of "JSON.namedheaders" (or "JSON-SEQ.namedheaders") for the "qlog_format" field (see Section 3).

The second option is to replace field values and/or names with indices into a (dynamic) lookup table. This is a common compression technique and can provide significant file size reductions (up to 50% in our tests, 100MB to 50MB). However, this approach is even more difficult to implement efficiently and requires either including the (dynamic) table in the resulting file (an approach taken by for example Chromium's NetLog format (<https://www.chromium.org/developers/design-documents/network-stack/netlog>)) or defining a (static) table up-front and sharing this between implementations. Implementations using this approach should not employ a separate file extension (as it still uses JSON), but rather employ a new value of "JSON.dictionary" (or "JSON-SEQ.dictionary") for the "qlog_format" field (see Section 3).

As both options either proved difficult to implement, reduced qlog file readability, and provided too little improvement compared to other more straightforward options (for example Section 6.3.2), these schemes are not inherently part of qlog.

6.3.2. Compression

The second general category of optimizations is to utilize a (generic) compression scheme for textual data. As qlog in the JSON(-SEQ) format typically contains a large amount of repetition, off-the-shelf (text) compression techniques typically succeed very well in bringing down file sizes (regularly with up to two orders of magnitude in our tests, even for "fast" compression levels). As such, utilizing compression is recommended before attempting other optimization options, even though this might (somewhat) increase processing costs due to the additional compression step.

The first option is to use GZIP compression ([RFC1952]). This generic compression scheme provides multiple compression levels (providing a trade-off between compression speed and size reduction). Utilized at level 6 (a medium setting thought to be applicable for streaming compression of a qlog stream in commodity devices), gzip compresses qlog JSON files to 7% of their initial size on average (100MB to 7MB). For this option, the file extension `.(s)qlog.gz` SHOULD BE used. The "qlog_format" field should still reflect the original JSON formatting of the qlog data (e.g., "JSON" or "JSON-SEQ").

The second option is to use Brotli compression ([RFC7932]). While similar to gzip, this more recent compression scheme provides a better efficiency. It also allows multiple compression levels. Utilized at level 4 (a medium setting thought to be applicable for streaming compression of a qlog stream in commodity devices), brotli compresses qlog JSON files to 7% of their initial size on average (100MB to 7MB). For this option, the file extension `.(s)qlog.br` SHOULD BE used. The "qlog_format" field should still reflect the original JSON formatting of the qlog data (e.g., "JSON" or "JSON-SEQ").

Other compression algorithms of course exist (for example xz, zstd, and lz4). We mainly recommend gzip and brotli because of their tweakable behaviour and wide support in web-based environments, which we envision as the main tooling ecosystem (see also Section 8).

6.3.3. Binary formats

The third general category of optimizations is to use a more optimized (often binary) format instead of the textual JSON format. This approach inherently produces smaller files and often has better (de)serialization performance. However, the resultant files are no longer human readable and some formats require hard tradeoffs between flexibility for performance.

The first option is to use the CBOR (Concise Binary Object Representation [RFC7049]) format. For our purposes, CBOR can be viewed as a straightforward binary variant of JSON. As such, existing JSON qlog files can be trivially converted to and from CBOR (though slightly more work is needed for JSON-SEQ qlogs to convert them to CBOR-SEQ, see [RFC8742]). While CBOR thus does retain the full qlog flexibility, it only provides a 25% file size reduction (100MB to 75MB) compared to textual JSON(-SEQ). As CBOR support in programming environments is not as widespread as that of textual JSON and the format lacks human readability, CBOR was not chosen as the default qlog format. For this option, the file extension `.(s)qlog.cbor` SHOULD BE used. The `"qlog_format"` field should still reflect the original JSON formatting of the qlog data (e.g., `"JSON"` or `"JSON-SEQ"`). The media type should indicate both whether JSON or JSON Text Sequences are used, as well as whether CBOR or CBOR Sequences are used (see the table below).

A second option is to use a more specialized binary format, such as Protocol Buffers (<https://developers.google.com/protocol-buffers>) (protobuf). This format is battle-tested, has support for optional fields and has libraries in most programming languages. Still, it is significantly less flexible than textual JSON or CBOR, as it relies on a separate, pre-defined schema (a `.proto` file). As such, it is not possible to (easily) log new event types in protobuf files without adjusting this schema as well, which has its own practical challenges. As qlog is intended to be a flexible, general purpose format, this type of format was not chosen as its basic serialization. The lower flexibility does lead to significantly reduced file sizes. Our straightforward mapping of the qlog main schema and QUIC/HTTP3 event types to protobuf created qlog files 24% as large as the raw JSON equivalents (100MB to 24MB). For this option, the file extension `.(s)qlog.protobuf` SHOULD BE used. The `"qlog_format"` field should reflect the different internal format, for example: `"qlog_format": "protobuf"`.

Note that binary formats can (and should) also be used in conjunction with compression (see Section 6.3.2). For example, CBOR compresses well (to about 6% of the original textual JSON size (100MB to 6MB) for both gzip and brotli) and so does protobuf (5% (gzip) to 3%

(brotli)). However, these gains are similar to the ones achieved by simply compression the textual JSON equivalents directly (7%, see Section 6.3.2). As such, since compression is still needed to achieve optimal file size reductions event with binary formats, we feel the more flexible compressed textual JSON options are a better default for the qlog format in general.

6.3.4. Overview and summary

In summary, textual JSON was chosen as the main qlog format due to its high flexibility and because its inefficiencies can be largely solved by the utilization of compression techniques (which are needed to achieve optimal results with other formats as well).

Still, qlog implementers are free to define other qlog formats depending on their needs and context of use. These formats should be described in their own documents, the discussion in this document mainly acting as inspiration and high-level guidance. Implementers are encouraged to add concrete qlog formats and definitions to the designated public repository (<https://github.com/quiclog/qlog>).

The following table provides an overview of all the discussed qlog formatting options with examples:

format	qlog_format	extension	media type
JSON Section 6.1	JSON	.qlog	application/ qlog+json
JSON Text Sequences Section 6.2	JSON-SEQ	.sqlog	application/ qlog+json- seq
named headers Section 6.3.1	JSON(- SEQ).namedheaders	.(s)qlog	application/ qlog+json(- seq)
dictionary Section 6.3.1	JSON(- SEQ).dictionary	.(s)qlog	application/ qlog+json(- seq)
CBOR Section 6.3.3	JSON(-SEQ)	.(s)qlog.cbor	application/ qlog+json(- seq)+cbor(- seq)
protobuf Section 6.3.3	protobuf	.qlog.protobuf	NOT SPECIFIED BY IANA
gzip Section 6.3.2	no change	.gz suffix	application/ gzip
brrotli Section 6.3.2	no change	.br suffix	NOT SPECIFIED BY IANA

Table 1

6.4. Conversion between formats

As discussed in the previous sections, a qlog file can be serialized in a multitude of formats, each of which can conceivably be transformed into or from one another without loss of information. For example, a number of JSON-SEQ streamed qlogs could be combined into a JSON formatted qlog for later processing. Similarly, a captured binary qlog could be transformed to JSON for easier

interpretation and sharing.

Secondly, we can also consider other structured logging approaches that contain similar (though typically not identical) data to qlog, like raw packet capture files (for example .pcap files from tcpdump) or endpoint-specific logging formats (for example the NetLog format in Google Chrome). These are sometimes the only options, if an implementation cannot or will not support direct qlog output for any reason, but does provide other internal or external (e.g., SSLKEYLOGFILE export to allow decryption of packet captures) logging options. For this second category, a (partial) transformation from/to qlog can also be defined.

As such, when defining a new qlog serialization format or wanting to utilize qlog-compatible tools with existing codebases lacking qlog support, it is recommended to define and provide a concrete mapping from one format to default JSON-serialized qlog. Several of such mappings exist. Firstly, [pcap2qlog] (<https://github.com/quiclog/pcap2qlog>) transforms QUIC and HTTP/3 packet capture files to qlog. Secondly, netlog2qlog (<https://github.com/quiclog/qvis/tree/master/visualizations/src/components/filemanager/netlogconverter>) converts chromium's internal dictionary-encoded JSON format to qlog. Finally, quictrace2qlog (<https://github.com/quiclog/quictrace2qlog>) converts the older quictrace format to JSON qlog. Tools can then easily integrate with these converters (either by incorporating them directly or for example using them as a (web-based) API) so users can provide different file types with ease. For example, the qvis (<https://qvis.edm.uhasselt.be>) toolsuite supports a multitude of formats and qlog serializations.

7. Methods of access and generation

Different implementations will have different ways of generating and storing qlogs. However, there is still value in defining a few default ways in which to steer this generation and access of the results.

7.1. Set file output destination via an environment variable

To provide users control over where and how qlog files are created, we define two environment variables. The first, QLOGFILE, indicates a full path to where an individual qlog file should be stored. This path MUST include the full file extension. The second, QLOGDIR, sets a general directory path in which qlog files should be placed. This path MUST include the directory separator character at the end.

In general, QLOGDIR should be preferred over QLOGFILE if an endpoint is prone to generate multiple qlog files. This can for example be the case for a QUIC server implementation that logs each QUIC connection in a separate qlog file. An alternative that uses QLOGFILE would be a QUIC server that logs all connections in a single file and uses the "group_id" field (Section 3.4.6) to allow post-hoc separation of events.

Implementations SHOULD provide support for QLOGDIR and MAY provide support for QLOGFILE.

When using QLOGDIR, it is up to the implementation to choose an appropriate naming scheme for the qlog files themselves. The chosen scheme will typically depend on the context or protocols used. For example, for QUIC, it is recommended to use the Original Destination Connection ID (ODCID), followed by the vantage point type of the logging endpoint. Examples of all options for QUIC are shown in Figure 41.

Command: QLOGFILE=/srv/qlogs/client.qlog quicclientbinary

Should result in the the quicclientbinary executable logging a single qlog file named client.qlog in the /srv/qlogs directory. This is for example useful in tests when the client sets up just a single connection and then exits.

Command: QLOGDIR=/srv/qlogs/ quicserverbinary

Should result in the quicserverbinary executable generating several logs files, one for each QUIC connection. Given two QUIC connections, with ODCID values "abcde" and "12345" respectively, this would result in two files:
/srv/qlogs/abcde_server.qlog
/srv/qlogs/12345_server.qlog

Command: QLOGFILE=/srv/qlogs/server.qlog quicserverbinary

Should result in the the quicserverbinary executable logging a single qlog file named server.qlog in the /srv/qlogs directory. Given that the server handled two QUIC connections before it was shut down, with ODCID values "abcde" and "12345" respectively, this would result in event instances in the qlog file being tagged with the "group_id" field with values "abcde" and "12345".

Figure 41: Environment variable examples for a QUIC implementation

7.2. Access logs via a well-known endpoint

After generation, qlog implementers MAY make available generated logs and traces on an endpoint (typically the server) via the following .well-known URI:

```
.well-known/qlog/IDENTIFIER.extension
```

The IDENTIFIER variable depends on the context and the protocol. For example for QUIC, the lowercase Original Destination Connection ID (ODCID) is recommended, as it can uniquely identify a connection. Additionally, the extension depends on the chosen format (see Section 6.3.4). For example, for a QUIC connection with ODCID "abcde", the endpoint for fetching its default JSON-formatted .qlog file would be:

```
.well-known/qlog/abcde.qlog
```

Implementers SHOULD allow users to fetch logs for a given connection on a 2nd, separate connection. This helps prevent pollution of the logs by fetching them over the same connection that one wishes to observe through the log. Ideally, for the QUIC use case, the logs should also be approachable via an HTTP/2 or HTTP/1.1 endpoint (i.e., on TCP port 443), to for example aid debugging in the case where QUIC/UDP is blocked on the network.

qlog implementers SHOULD NOT enable this .well-known endpoint in typical production settings to prevent (malicious) users from downloading logs from other connections. Implementers are advised to disable this endpoint by default and require specific actions from the end users to enable it (and potentially qlog itself). Implementers MUST also take into account the general privacy and security guidelines discussed in Section 9 before exposing qlogs to outside actors.

8. Tooling requirements

Tools ingestion qlog MUST indicate which qlog version(s), qlog format(s), compression methods and potentially other input file formats (for example .pcap) they support. Tools SHOULD at least support .qlog files in the default JSON format (Section 6.1). Additionally, they SHOULD indicate exactly which values for and properties of the name (category and type) and data fields they look for to execute their logic. Tools SHOULD perform a (high-level) check if an input qlog file adheres to the expected qlog schema. If a tool determines a qlog file does not contain enough supported information to correctly execute the tool's logic, it SHOULD generate a clear error message to this effect.

Tools MUST NOT produce breaking errors for any field names and/or values in the qlog format that they do not recognize. Tools SHOULD indicate even unknown event occurrences within their context (e.g., marking unknown events on a timeline for manual interpretation by the user).

Tool authors should be aware that, depending on the logging implementation, some events will not always be present in all traces. For example, using a circular logging buffer of a fixed size, it could be that the earliest events (e.g., connection setup events) are later overwritten by "newer" events. Alternatively, some events can be intentionally omitted out of privacy or file size considerations. Tool authors are encouraged to make their tools robust enough to still provide adequate output for incomplete logs.

9. Security and privacy considerations

TODO : discuss privacy and security considerations (e.g., what NOT to log, what to strip out of a log before sharing, ...)

TODO: strip out/don't log IPs, ports, specific CIDs, raw user data, exact times, HTTP HEADERS (or at least :path), SNI values

TODO: see if there is merit in encrypting the logs and having the server choose an encryption key (e.g., sent in transport parameters)

Good initial reference: Christian Huitema's blogpost
(<https://huitema.wordpress.com/2020/07/21/scrubbing-quic-logs-for-privacy/>)

10. IANA Considerations

TODO: primarily the .well-known URI

11. References

11.1. Normative References

- [CDDL] Birkholz, H., Vigano, C., and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures", RFC 8610, DOI 10.17487/RFC8610, June 2019, <<https://www.rfc-editor.org/rfc/rfc8610>>.
- [I-JSON] Bray, T., Ed., "The I-JSON Message Format", RFC 7493, DOI 10.17487/RFC7493, March 2015, <<https://www.rfc-editor.org/rfc/rfc7493>>.

- [JSON] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <<https://www.rfc-editor.org/rfc/rfc8259>>.
- [JSON-Text-Sequences] Williams, N., "JavaScript Object Notation (JSON) Text Sequences", RFC 7464, DOI 10.17487/RFC7464, February 2015, <<https://www.rfc-editor.org/rfc/rfc7464>>.
- [QLOG-H3] Marx, R., Ed., Niccolini, L., Ed., and M. Seemann, Ed., "HTTP/3 and QPACK event definitions for qlog", Work in Progress, Internet-Draft, draft-ietf-quic-qlog-h3-events-01, <<https://datatracker.ietf.org/doc/html/draft-ietf-quic-qlog-h3-events-01>>.
- [QLOG-QUIC] Marx, R., Ed., Niccolini, L., Ed., and M. Seemann, Ed., "QUIC event definitions for qlog", Work in Progress, Internet-Draft, draft-ietf-quic-qlog-quic-events-01, <<https://datatracker.ietf.org/doc/html/draft-ietf-quic-qlog-quic-events-01>>.
- [RFC1952] Deutsch, P., "GZIP file format specification version 4.3", RFC 1952, DOI 10.17487/RFC1952, May 1996, <<https://www.rfc-editor.org/rfc/rfc1952>>.
- [RFC4180] Shafranovich, Y., "Common Format and MIME Type for Comma-Separated Values (CSV) Files", RFC 4180, DOI 10.17487/RFC4180, October 2005, <<https://www.rfc-editor.org/rfc/rfc4180>>.
- [RFC6839] Hansen, T. and A. Melnikov, "Additional Media Type Structured Syntax Suffixes", RFC 6839, DOI 10.17487/RFC6839, January 2013, <<https://www.rfc-editor.org/rfc/rfc6839>>.
- [RFC7049] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", RFC 7049, DOI 10.17487/RFC7049, October 2013, <<https://www.rfc-editor.org/rfc/rfc7049>>.
- [RFC7464] Williams, N., "JavaScript Object Notation (JSON) Text Sequences", RFC 7464, DOI 10.17487/RFC7464, February 2015, <<https://www.rfc-editor.org/rfc/rfc7464>>.
- [RFC7932] Alakuijala, J. and Z. Szabadka, "Brotli Compressed Data Format", RFC 7932, DOI 10.17487/RFC7932, July 2016, <<https://www.rfc-editor.org/rfc/rfc7932>>.

- [RFC8091] Wilde, E., "A Media Type Structured Syntax Suffix for JSON Text Sequences", RFC 8091, DOI 10.17487/RFC8091, February 2017, <<https://www.rfc-editor.org/rfc/rfc8091>>.
- [RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <<https://www.rfc-editor.org/rfc/rfc8259>>.

11.2. Informative References

- [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/rfc/rfc2119>>.
- [RFC8742] Bormann, C., "Concise Binary Object Representation (CBOR) Sequences", RFC 8742, DOI 10.17487/RFC8742, February 2020, <<https://www.rfc-editor.org/rfc/rfc8742>>.

Appendix A. Change Log

A.1. Since draft-ietf-quic-qlog-main-schema-01:

- * Change the data definition language from TypeScript to CDDL (#143)

A.2. Since draft-ietf-quic-qlog-main-schema-00:

- * Changed the streaming serialization format from NDJSON to JSON Text Sequences (#172)
- * Added Media Type definitions for various qlog formats (#158)
- * Changed to semantic versioning

A.3. Since draft-marx-qlog-main-schema-draft-02:

- * These changes were done in preparation of the adoption of the drafts by the QUIC working group (#137)
- * Moved RawInfo, Importance, Generic events and Simulation events to this document.
- * Added basic event definition guidelines
- * Made protocol_type an array instead of a string (#146)

A.4. Since draft-marx-qlog-main-schema-01:

- * Decoupled qlog from the JSON format and described a mapping instead (#89)
 - Data types are now specified in this document and proper definitions for fields were added in this format
 - 64-bit numbers can now be either strings or numbers, with a preference for numbers (#10)
 - binary blobs are now logged as lowercase hex strings (#39, #36)
 - added guidance to add length-specifiers for binary blobs (#102)
- * Removed "time_units" from Configuration. All times are now in ms instead (#95)
- * Removed the "event_fields" setup for a more straightforward JSON format (#101, #89)
- * Added a streaming option using the NDJSON format (#109, #2, #106)
- * Described optional optimization options for implementers (#30)
- * Added QLOGDIR and QLOGFILE environment variables, clarified the .well-known URL usage (#26, #33, #51)
- * Overall tightened up the text and added more examples

A.5. Since draft-marx-qlog-main-schema-00:

- * All field names are now lowercase (e.g., category instead of CATEGORY)
- * Triggers are now properties on the "data" field value, instead of separate field types (#23)
- * group_ids in common_fields is now just also group_id

Appendix B. Design Variations

- * Quic-trace (<https://github.com/google/quic-trace>) takes a slightly different approach based on protocolbuffers.
- * Spindump (<https://github.com/EricssonResearch/spindump>) also defines a custom text-based format for in-network measurements

- * Wireshark (<https://www.wireshark.org/>) also has a QUIC dissector and its results can be transformed into a json output format using tshark.

The idea is that qlog is able to encompass the use cases for both of these alternate designs and that all tooling converges on the qlog standard.

Appendix C. Acknowledgements

Much of the initial work by Robin Marx was done at Hasselt University.

Thanks to Jana Iyengar, Brian Trammell, Dmitri Tikhonov, Stephen Petrides, Jari Arkko, Marcus Ihlar, Victor Vasiliev, Mirja Kuehlewind, Jeremy Laine and Lucas Pardue for their feedback and suggestions.

Authors' Addresses

Robin Marx (editor)
KU Leuven
Email: robin.marx@kuleuven.be

Luca Niccolini (editor)
Facebook
Email: lniccolini@fb.com

Marten Seemann (editor)
Protocol Labs
Email: marten@protocol.ai

QUIC
Internet-Draft
Intended status: Standards Track
Expires: 8 September 2022

R. Marx
KU Leuven
L. Niccolini, Ed.
Facebook
M. Seemann, Ed.
Protocol Labs
7 March 2022

QUIC event definitions for qlog
draft-ietf-quic-qlog-quic-events-01

Abstract

This document describes concrete qlog event definitions and their metadata for QUIC events. These events can then be embedded in the higher level schema defined in [QLOG-MAIN].

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 September 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction	4
1.1. Notational Conventions	4
2. Overview	4
2.1. Links to the main schema	5
2.1.1. Raw packet and frame information	5
2.1.2. Events not belonging to a single connection	6
3. QUIC event definitions	6
3.1. connectivity	7
3.1.1. server_listening	7
3.1.2. connection_started	7
3.1.3. connection_closed	8
3.1.4. connection_id_updated	9
3.1.5. spin_bit_updated	10
3.1.6. connection_retried	10
3.1.7. connection_state_updated	10
3.1.8. MIGRATION-related events	13
3.2. security	13
3.2.1. key_updated	13
3.2.2. key_retired	14
3.3. transport	14
3.3.1. version_information	14
3.3.2. alpn_information	15
3.3.3. parameters_set	16
3.3.4. parameters_restored	18
3.3.5. packet_sent	18
3.3.6. packet_received	19
3.3.7. packet_dropped	20
3.3.8. packet_buffered	21
3.3.9. packets_acked	22
3.3.10. datagrams_sent	23
3.3.11. datagrams_received	23
3.3.12. datagram_dropped	24
3.3.13. stream_state_updated	24
3.3.14. frames_processed	26
3.3.15. data_moved	27
3.4. recovery	28
3.4.1. parameters_set	28
3.4.2. metrics_updated	29
3.4.3. congestion_state_updated	30
3.4.4. loss_timer_updated	31
3.4.5. packet_lost	32
3.4.6. marked_for_retransmit	33
4. Security Considerations	33
5. IANA Considerations	33
6. References	33
6.1. Normative References	33

6.2. Informative References	34
Appendix A. QUIC data field definitions	34
A.1. ProtocolEventBody extension	34
A.2. QuicVersion	35
A.3. ConnectionID	35
A.4. Owner	35
A.5. IPAddress and IPVersion	35
A.6. PacketType	36
A.7. PacketNumberSpace	36
A.8. PacketHeader	36
A.9. Token	37
A.10. KeyType	37
A.11. QUIC Frames	37
A.11.1. PaddingFrame	38
A.11.2. PingFrame	38
A.11.3. AckFrame	38
A.11.4. ResetStreamFrame	39
A.11.5. StopSendingFrame	40
A.11.6. CryptoFrame	40
A.11.7. NewTokenFrame	40
A.11.8. StreamFrame	41
A.11.9. MaxDataFrame	41
A.11.10. MaxStreamDataFrame	41
A.11.11. MaxStreamsFrame	42
A.11.12. DataBlockedFrame	42
A.11.13. StreamDataBlockedFrame	42
A.11.14. StreamsBlockedFrame	42
A.11.15. NewConnectionIDFrame	42
A.11.16. RetireConnectionIDFrame	43
A.11.17. PathChallengeFrame	43
A.11.18. PathResponseFrame	43
A.11.19. ConnectionCloseFrame	44
A.11.20. HandshakeDoneFrame	44
A.11.21. UnknownFrame	44
A.11.22. TransportError	44
A.11.23. ApplicationError	45
A.11.24. CryptoError	45
Appendix B. Change Log	45
B.1. Since draft-ietf-qlog-quic-events-00:	45
B.2. Since draft-marx-qlog-event-definitions-quic-h3-02:	46
B.3. Since draft-marx-qlog-event-definitions-quic-h3-01:	46
B.4. Since draft-marx-qlog-event-definitions-quic-h3-00:	47
Appendix C. Design Variations	48
Appendix D. Acknowledgements	48
Authors' Addresses	48

1. Introduction

This document describes the values of the qlog name ("category" + "event") and "data" fields and their semantics for the QUIC protocol. This document is based on draft-34 of the QUIC I-Ds [QUIC-TRANSPORT], [QUIC-RECOVERY], and [QUIC-TLS]. HTTP/3 and QPACK events are defined in a separate document [QLOG-H3].

Feedback and discussion are welcome at <https://github.com/quicwg/qlog> (<https://github.com/quicwg/qlog>). Readers are advised to refer to the "editor's draft" at that URL for an up-to-date version of this document.

Concrete examples of integrations of this schema in various programming languages can be found at <https://github.com/quiclog/qlog/> (<https://github.com/quiclog/qlog/>).

1.1. Notational Conventions

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 [RFC2119].

The event and data structure definitions in this document are expressed in the Concise Data Definition Language [CDDL] and its extensions described in [QLOG-MAIN].

2. Overview

This document describes the values of the qlog "name" ("category" + "event") and "data" fields and their semantics for the QUIC protocol.

This document assumes the usage of the encompassing main qlog schema defined in [QLOG-MAIN]. Each subsection below defines a separate category (for example connectivity, transport, recovery) and each subsubsection is an event type (for example packet_received).

For each event type, its importance and data definition is laid out, often accompanied by possible values for the optional "trigger" field. For the definition and semantics of "importance" and "trigger", see the main schema document.

Most of the complex datastructures, enums and re-usable definitions are grouped together on the bottom of this document for clarity.

2.1. Links to the main schema

This document re-uses all the fields defined in the main qlog schema (e.g., name, category, type, data, group_id, protocol_type, the time-related fields, importance, RawInfo, etc.).

One entry in the "protocol_type" qlog array field MUST be "QUIC" if events from this document are included in a qlog trace.

When the qlog "group_id" field is used, it is recommended to use QUIC's Original Destination Connection ID (ODCID, the CID chosen by the client when first contacting the server), as this is the only value that does not change over the course of the connection and can be used to link more advanced QUIC packets (e.g., Retry, Version Negotiation) to a given connection. Similarly, the ODCID should be used as the qlog filename or file identifier, potentially suffixed by the vantagepoint type (For example, abcd1234_server.qlog would contain the server-side trace of the connection with ODCID abcd1234).

2.1.1. Raw packet and frame information

This document re-uses the definition of the RawInfo data class from [QLOG-MAIN].

Note: QUIC packets always include an AEAD authentication tag ("trailer") at the end. As this tag is always the same size for a given connection (it depends on the used TLS cipher), this document does not define a separate "RawInfo:aead_tag_length" field here. Instead, this field is reflected in "transport:parameters_set" and can be logged only once.

Note: As QUIC uses trailers in packets, packet header_lengths can be calculated as:

$$\text{header_length} = \text{length} - \text{payload_length} - \text{aead_tag_length}$$

For UDP datagrams, the calculation is simpler:

$$\text{header_length} = \text{length} - \text{payload_length}$$

Note: In some cases, the length fields are also explicitly reflected

inside of packet headers. For example, the QUIC STREAM frame has a "length" field indicating its payload size. Similarly, the QUIC Long Header has a "length" field which is equal to the payload length plus the packet number length. In these cases, those fields are intentionally preserved in the event definitions. Even though this can lead to duplicate data when the full RawInfo is logged, it allows a more direct mapping of the QUIC specifications to qlog, making it easier for users to interpret.

2.1.2. Events not belonging to a single connection

For several types of events, it is sometimes impossible to tie them to a specific conceptual QUIC connection (e.g., a `packet_dropped` event triggered because the packet has an unknown `connection_id` in the header). Since qlog events in a trace are typically associated with a single connection, it is unclear how to log these events.

Ideally, implementers SHOULD create a separate, individual "endpoint-level" trace file (or `group_id` value), not associated with a specific connection (for example a "server.qlog" or `group_id = "client"`), and log all events that do not belong to a single connection to this grouping trace. However, this is not always practical, depending on the implementation. Because the semantics of most of these events are well-defined in the protocols and because they are difficult to mis-interpret as belonging to a connection, implementers MAY choose to log events not belonging to a particular connection in any other trace, even those strongly associated with a single connection.

Note that this can make it difficult to match logs from different vantage points with each other. For example, from the client side, it is easy to log connections with version negotiation or retry in the same trace, while on the server they would most likely be logged in separate traces. Servers can take extra efforts (and keep additional state) to keep these events combined in a single trace however (for example by also matching connections on their four-tuple instead of just the connection ID).

3. QUIC event definitions

Each subheading in this section is a qlog event category, while each sub-subheading is a qlog event type. Concretely, for the following two items, we have the category "connectivity" and event type "server_listening", resulting in a concatenated qlog "name" field value of "connectivity:server_listening".

3.1. connectivity

3.1.1. server_listening

Importance: Extra

Emitted when the server starts accepting connections.

Definition:

```
ConnectivityServerListening = {  
  ? ip_v4: IPAddress  
  ? ip_v6: IPAddress  
  ? port_v4: uint16  
  ? port_v6: uint16  
  
  ; the server will always answer client initials with a retry  
  ; (no 1-RTT connection setups by choice)  
  ? retry_required: bool  
}
```

Figure 1: ConnectivityServerListening definition

Note: some QUIC stacks do not handle sockets directly and are thus unable to log IP and/or port information.

3.1.2. connection_started

Importance: Base

Used for both attempting (client-perspective) and accepting (server-perspective) new connections. Note that this event has overlap with `connection_state_updated` and this is a separate event mainly because of all the additional data that should be logged.

Definition:

```
ConnectivityConnectionStarted = {  
  ? ip_version: IPVersion  
  src_ip: IPAddress  
  dst_ip: IPAddress  
  
  ; transport layer protocol  
  ? protocol: text .default "QUIC"  
  ? src_port: uint16  
  ? dst_port: uint16  
  
  ? src_cid: ConnectionID  
  ? dst_cid: ConnectionID  
}
```

Figure 2: ConnectivityConnectionStarted definition

Note: some QUIC stacks do not handle sockets directly and are thus unable to log IP and/or port information.

3.1.3. connection_closed

Importance: Base

Used for logging when a connection was closed, typically when an error or timeout occurred. Note that this event has overlap with `connectivity:connection_state_updated`, as well as the `CONNECTION_CLOSE` frame. However, in practice, when analyzing large deployments, it can be useful to have a single event representing a `connection_closed` event, which also includes an additional reason field to provide additional information. Additionally, it is useful to log closures due to timeouts, which are difficult to reflect using the other options.

In QUIC there are two main connection-closing error categories: connection and application errors. They have well-defined error codes and semantics. Next to these however, there can be internal errors that occur that may or may not get mapped to the official error codes in implementation-specific ways. As such, multiple error codes can be set on the same event to reflect this.

Definition:

```

ConnectivityConnectionClosed = {
  ; which side closed the connection
  ? owner: Owner

  ? connection_code: TransportError / CryptoError / uint32
  ? application_code: $ApplicationError / uint32
  ? internal_code: uint32

  ? reason: text
  ? trigger:
    "clean" /
    "handshake_timeout" /
    "idle_timeout" /
    ; this is called the "immediate close" in the QUIC RFC
    "error" /
    "stateless_reset" /
    "version_mismatch" /
    ; for example HTTP/3's GOAWAY frame
    "application"
}

```

Figure 3: ConnectivityConnectionClosed definition

3.1.4. connection_id_updated

Importance: Base

This event is emitted when either party updates their current Connection ID. As this typically happens only sparingly over the course of a connection, this event allows loggers to be more efficient than logging the observed CID with each packet in the .header field of the "packet_sent" or "packet_received" events.

This is viewed from the perspective of the one applying the new id. As such, if we receive a new connection id from our peer, we will see the dst_ fields are set. If we update our own connection id (e.g., NEW_CONNECTION_ID frame), we log the src_ fields.

Definition:

```

ConnectivityConnectionIDUpdated = {
  owner: Owner

  ? old: ConnectionID
  ? new: ConnectionID
}

```

Figure 4: ConnectivityConnectionIDUpdated definition

3.1.5. spin_bit_updated

Importance: Base

To be emitted when the spin bit changes value. It SHOULD NOT be emitted if the spin bit is set without changing its value.

Definition:

```
ConnectivitySpinBitUpdated = {  
    state: bool  
}
```

Figure 5: ConnectivitySpinBitUpdated definition

3.1.6. connection_retried

TODO

3.1.7. connection_state_updated

Importance: Base

This event is used to track progress through QUIC's complex handshake and connection close procedures. It is intended to provide exhaustive options to log each state individually, but also provides a more basic, simpler set for implementations less interested in tracking each smaller state transition. As such, users should not expect to see -all- these states reflected in all qlogs and implementers should focus on support for the SimpleConnectionState set.

Definition:

```

ConnectivityConnectionStateUpdated = {
  ? old: ConnectionState / SimpleConnectionState
  new: ConnectionState / SimpleConnectionState
}

ConnectionState =
  ; initial sent/received
  "attempted" /
  ; peer address validated by: client sent Handshake packet OR
  ; client used CONNID chosen by the server.
  ; transport-draft-32, section-8.1
  "peer_validated" /
  "handshake_started" /
  ; 1 RTT can be sent, but handshake isn't done yet
  "early_write" /
  ; TLS handshake complete: Finished received and sent
  ; tls-draft-32, section-4.1.1
  "handshake_complete" /
  ; HANDSHAKE_DONE sent/received (connection is now "active", 1RTT
  ; can be sent). tls-draft-32, section-4.1.2
  "handshake_confirmed" /
  "closing" /
  ; connection_close sent/received
  "draining" /
  ; draining period done, connection state discarded
  "closed"

SimpleConnectionState =
  "attempted" /
  "handshake_started" /
  "handshake_confirmed" /
  "closed"

```

Figure 6: ConnectivityConnectionStateUpdated definition

These states correspond to the following transitions for both client and server:

Client:

* send initial

- state = attempted

* get initial

- state = validated _(not really "needed" at the client, but somewhat useful to indicate progress nonetheless)_

```
* get first Handshake packet
  - state = handshake_started

* get Handshake packet containing ServerFinished
  - state = handshake_complete

* send ClientFinished
  - state = early_write (1RTT can now be sent)

* get HANDSHAKE_DONE
  - state = handshake_confirmed

*Server:*

* get initial
  - state = attempted

* send initial _(TODO don't think this needs a separate state, since
  some handshake will always be sent in the same flight as this?)_

* send handshake EE, CERT, CV, ...
  - state = handshake_started

* send ServerFinished
  - state = early_write (1RTT can now be sent)

* get first handshake packet / something using a server-issued CID
  of min length
  - state = validated

* get handshake packet containing ClientFinished
  - state = handshake_complete

* send HANDSHAKE_DONE
  - state = handshake_confirmed
```

Note: connection_state_changed with a new state of "attempted" is

the same conceptual event as the `connection_started` event above from the client's perspective. Similarly, a state of "closing" or "draining" corresponds to the `connection_closed` event.

3.1.8. MIGRATION-related events

e.g., `path_updated`

TODO: read up on the draft how migration works and whether to best fit this here or in TRANSPORT TODO: integrate
<https://tools.ietf.org/html/draft-deconinck-quic-multipath-02>

For now, infer from other connectivity events and `path_challenge/`
`path_response` frames

3.2. security

3.2.1. key_updated

Importance: Base

Note: `secret_updated` would be more correct, but in the draft it's called `KEY_UPDATE`, so stick with that for consistency

Definition:

```
SecurityKeyUpdated = {  
    key_type: KeyType  
  
    ? old: hexstring  
    new: hexstring  
  
    ; needed for 1RTT key updates  
    ? generation: uint32  
  
    ? trigger:  
        ; (e.g., initial, handshake and 0-RTT keys  
        ; are generated by TLS)  
        "tls" /  
        "remote_update" /  
        "local_update"  
}
```

Figure 7: SecurityKeyUpdated definition

3.2.2. key_retired

Importance: Base

Definition:

```
SecurityKeyRetired = {  
  key_type: KeyType  
  ? key: hexstring  
  
  ; needed for 1RTT key updates  
  ? generation: uint32  
  
  ? trigger:  
    ; (e.g., initial, handshake and 0-RTT keys  
    ; are generated by TLS)  
    "tls" /  
    "remote_update" /  
    "local_update"  
}
```

Figure 8: SecurityKeyRetired definition

3.3. transport

3.3.1. version_information

Importance: Core

QUIC endpoints each have their own list of of QUIC versions they support. The client uses the most likely version in their first initial. If the server does support that version, it replies with a version_negotiation packet, containing supported versions. From this, the client selects a version. This event aggregates all this information in a single event type. It also allows logging of supported versions at an endpoint without actual version negotiation needing to happen.

Definition:

```
TransportVersionInformation = {  
  ? server_versions: [+ QuicVersion]  
  ? client_versions: [+ QuicVersion]  
  ? chosen_version: QuicVersion  
}
```

Figure 9: TransportVersionInformation definition

Intended use:

- * When sending an initial, the client logs this event with `client_versions` and `chosen_version` set
- * Upon receiving a client initial with a supported version, the server logs this event with `server_versions` and `chosen_version` set
- * Upon receiving a client initial with an unsupported version, the server logs this event with `server_versions` set and `client_versions` to the single-element array containing the client's attempted version. The absence of `chosen_version` implies no overlap was found.
- * Upon receiving a version negotiation packet from the server, the client logs this event with `client_versions` set and `server_versions` to the versions in the version negotiation packet and `chosen_version` to the version it will use for the next initial packet

3.3.2. alpn_information

Importance: Core

QUIC implementations each have their own list of application level protocols and versions thereof they support. The client includes a list of their supported options in its first initial as part of the TLS Application Layer Protocol Negotiation (alpn) extension. If there are common option(s), the server chooses the most optimal one and communicates this back to the client. If not, the connection is closed.

Definition:

```
TransportALPNInformation = {  
    ? server_alpns: [* text]  
    ? client_alpns: [* text]  
    ? chosen_alpn: text  
}
```

Figure 10: TransportALPNInformation definition

Intended use:

- * When sending an initial, the client logs this event with `client_alpns` set

- * When receiving an initial with a supported alpn, the server logs this event with `server_alpns` set, `client_alpns` equalling the client-provided list, and `chosen_alpn` to the value it will send back to the client.
- * When receiving an initial with an alpn, the client logs this event with `chosen_alpn` to the received value.
- * Alternatively, a client can choose to not log the first event, but wait for the receipt of the server initial to log this event with both `client_alpns` and `chosen_alpn` set.

3.3.3. `parameters_set`

Importance: Core

This event groups settings from several different sources (transport parameters, TLS ciphers, etc.) into a single event. This is done to minimize the amount of events and to decouple conceptual setting impacts from their underlying mechanism for easier high-level reasoning.

All these settings are typically set once and never change. However, they are typically set at different times during the connection, so there will typically be several instances of this event with different fields set.

Note that some settings have two variations (one set locally, one requested by the remote peer). This is reflected in the "owner" field. As such, this field **MUST** be correct for all settings included a single event instance. If you need to log settings from two sides, you **MUST** emit two separate event instances.

In the case of connection resumption and 0-RTT, some of the server's parameters are stored up-front at the client and used for the initial connection startup. They are later updated with the server's reply. In these cases, utilize the separate `parameters_restored` event to indicate the initial values, and this event to indicate the updated values, as normal.

Definition:

```
TransportParametersSet = {  
  ? owner: Owner  
  
  ; true if valid session ticket was received  
  ? resumption_allowed: bool
```

```
; true if early data extension was enabled on the TLS layer
? early_data_enabled: bool

; e.g., "AES_128_GCM_SHA256"
? tls_cipher: text

; depends on the TLS cipher, but it's easier to be explicit.
; in bytes
? aead_tag_length: uint8 .default 16

; transport parameters from the TLS layer:
? original_destination_connection_id: ConnectionID
? initial_source_connection_id: ConnectionID
? retry_source_connection_id: ConnectionID
? stateless_reset_token: Token
? disable_active_migration: bool

? max_idle_timeout: uint64
? max_udp_payload_size: uint32
? ack_delay_exponent: uint16
? max_ack_delay: uint16
? active_connection_id_limit: uint32

? initial_max_data: uint64
? initial_max_stream_data_bidi_local: uint64
? initial_max_stream_data_bidi_remote: uint64
? initial_max_stream_data_uni: uint64
? initial_max_streams_bidi: uint64
? initial_max_streams_uni: uint64

? preferred_address: PreferredAddress
}

PreferredAddress = {
  ip_v4: IPAddress
  ip_v6: IPAddress

  port_v4: uint16
  port_v6: uint16

  connection_id: ConnectionID
  stateless_reset_token: Token
}
```

Figure 11: TransportParametersSet definition

Additionally, this event can contain any number of unspecified fields. This is to reflect setting of for example unknown (greased) transport parameters or employed (proprietary) extensions.

3.3.4. parameters_restored

Importance: Base

When using QUIC 0-RTT, clients are expected to remember and restore the server's transport parameters from the previous connection. This event is used to indicate which parameters were restored and to which values when utilizing 0-RTT. Note that not all transport parameters should be restored (many are even prohibited from being re-utilized). The ones listed here are the ones expected to be useful for correct 0-RTT usage.

Definition:

```
TransportParametersRestored = {  
  ? disable_active_migration: bool  
  
  ? max_idle_timeout: uint64  
  ? max_udp_payload_size: uint32  
  ? active_connection_id_limit: uint32  
  
  ? initial_max_data: uint64  
  ? initial_max_stream_data_bidi_local: uint64  
  ? initial_max_stream_data_bidi_remote: uint64,  
  ? initial_max_stream_data_uni: uint64  
  ? initial_max_streams_bidi: uint64  
  ? initial_max_streams_uni: uint64  
}
```

Figure 12: TransportParametersRestored definition

Note that, like parameters_set above, this event can contain any number of unspecified fields to allow for additional/custom parameters.

3.3.5. packet_sent

Importance: Core

Definition:

```

TransportPacketSent = {
  header: PacketHeader

  ; see appendix for the QuicFrame definitions
  ? frames: [* QuicFrame]

  ? is_coalesced: bool .default false

  ; only if header.packet_type === "retry"
  ? retry_token: Token

  ; only if header.packet_type === "stateless_reset"
  ; is always 128 bits in length.
  ? stateless_reset_token: hexstring .size 16

  ; only if header.packet_type === "version_negotiation"
  ? supported_versions: [+ QuicVersion]

  ? raw: RawInfo
  ? datagram_id: uint32

  ? trigger:
    ; draft-23 5.1.1
    "retransmit_reordered" /
    ; draft-23 5.1.2
    "retransmit_timeout" /
    ; draft-23 5.3.1
    "pto_probe" /
    ; draft-19 6.2
    "retransmit_crypto" /
    ; needed for some CCs to figure out bandwidth allocations
    ; when there are no normal sends
    "cc_bandwidth_probe"
}

```

Figure 13: TransportPacketSent definition

Note: We do not explicitly log the `encryption_level` or `packet_number_space`: the `header.packet_type` specifies this by inference (assuming correct implementation)

Note: for more details on "datagram_id", see Section 3.3.10. It is only needed when keeping track of packet coalescing.

3.3.6. packet_received

Importance: Core

Definition:

```
TransportPacketReceived = {
  header: PacketHeader

  ; see appendix for the definitions
  ? frames: [* QuicFrame]

  ? is_coalesced: bool .default false

  ; only if header.packet_type === "retry"
  ? retry_token: Token

  ; only if header.packet_type === "stateless_reset"
  ; Is always 128 bits in length.
  ? stateless_reset_token: hexstring .size 16

  ; only if header.packet_type === "version_negotiation"
  ? supported_versions: [+ QuicVersion]

  ? raw: RawInfo
  ? datagram_id: uint32

  ? trigger:
    ; if packet was buffered because
    ; it couldn't be decrypted before
    "keys_available"
}
```

Figure 14: TransportPacketReceived definition

Note: We do not explicitly log the `encryption_level` or `packet_number_space`: the `header.packet_type` specifies this by inference (assuming correct implementation)

Note: for more details on "datagram_id", see Section 3.3.10. It is only needed when keeping track of packet coalescing.

3.3.7. packet_dropped

Importance: Base

This event indicates a QUIC-level packet was dropped after partial or no parsing.

Definition:


```
TransportPacketDropped = {  
    ; primarily packet_type should be filled here,  
    ; as other fields might not be parseable  
    ? header: PacketHeader  
  
    ? raw: RawInfo  
    ? datagram_id: uint32  
  
    ? trigger:  
        "key_unavailable" /  
        "unknown_connection_id" /  
        "header_parse_error" /  
        "payload_decrypt_error" /  
        "protocol_violation" /  
        "dos_prevention" /  
        "unsupported_version" /  
        "unexpected_packet" /  
        "unexpected_source_connection_id" /  
        "unexpected_version" /  
        "duplicate" /  
        "invalid_initial"  
}
```

Figure 15: TransportPacketDropped definition

Note: sometimes packets are dropped before they can be associated with a particular connection (e.g., in case of "unsupported_version"). This situation is discussed more in Section 2.1.2.

Note: for more details on "datagram_id", see Section 3.3.10. It is only needed when keeping track of packet coalescing.

3.3.8. packet_buffered

Importance: Base

This event is emitted when a packet is buffered because it cannot be processed yet. Typically, this is because the packet cannot be parsed yet, and thus we only log the full packet contents when it was parsed in a packet_received event.

Definition:

```
TransportPacketBuffered = {  
    ; primarily packet_type and possible packet_number should be  
    ; filled here as other elements might not be available yet  
    ? header: PacketHeader  
  
    ? raw: RawInfo  
    ? datagram_id: uint32  
  
    ? trigger:  
        ; indicates the parser cannot keep up, temporarily buffers  
        ; packet for later processing  
        "backpressure" /  
        ; if packet cannot be decrypted because the proper keys were  
        ; not yet available  
        "keys_unavailable"  
}
```

Figure 16: TransportPacketBuffered definition

Note: for more details on "datagram_id", see Section 3.3.10. It is only needed when keeping track of packet coalescing.

3.3.9. packets_acked

Importance: Extra

This event is emitted when a (group of) sent packet(s) is acknowledged by the remote peer for the first time. This information could also be deduced from the contents of received ACK frames. However, ACK frames require additional processing logic to determine when a given packet is acknowledged for the first time, as QUIC uses ACK ranges which can include repeated ACKs. Additionally, this event can be used by implementations that do not log frame contents.

Definition:

```
TransportPacketsAacked = {  
    ? packet_number_space: PacketNumberSpace  
  
    ? packet_numbers: [+ uint64]  
}
```

Figure 17: TransportPacketsAacked definition

Note: if packet_number_space is omitted, it assumes the default value of PacketNumberSpace.application_data, as this is by far the most prevalent packet number space a typical QUIC connection will use.

3.3.10. datagrams_sent

Importance: Extra

When we pass one or more UDP-level datagrams to the socket. This is useful for determining how QUIC packet buffers are drained to the OS.

Definition:

```
TransportDatagramsSent = {  
    ; to support passing multiple at once  
    ? count: uint16  
  
    ; RawInfo:length field indicates total length of the datagrams  
    ; including UDP header length  
    ? raw: [+ RawInfo]  
  
    ? datagram_ids: [+ uint32]  
}
```

Figure 18: TransportDatagramsSent definition

Note: QUIC itself does not have a concept of a "datagram_id". This field is a purely qlog-specific construct to allow tracking how multiple QUIC packets are coalesced inside of a single UDP datagram, which is an important optimization during the QUIC handshake. For this, implementations assign a (per-endpoint) unique ID to each datagram and keep track of which packets were coalesced into the same datagram. As packet coalescing typically only happens during the handshake (as it requires at least one long header packet), this can be done without much overhead.

3.3.11. datagrams_received

Importance: Extra

When we receive one or more UDP-level datagrams from the socket. This is useful for determining how datagrams are passed to the user space stack from the OS.

Definition:

```
TransportDatagramsReceived = {  
  ; to support passing multiple at once  
  ? count: uint16  
  
  ; RawInfo:length field indicates total length of the datagrams  
  ; including UDP header length  
  ? raw: [+ RawInfo]  
  
  ? datagram_ids: [+ uint32]  
}
```

Figure 19: TransportDatagramsReceived definition

Note: for more details on "datagram_ids", see Section 3.3.10.

3.3.12. datagram_dropped

Importance: Extra

When we drop a UDP-level datagram. This is typically if it does not contain a valid QUIC packet (in that case, use packet_dropped instead).

Definition:

```
TransportDatagramDropped = {  
  ? raw: RawInfo  
}
```

Figure 20: TransportDatagramDropped definition

3.3.13. stream_state_updated

Importance: Base

This event is emitted whenever the internal state of a QUIC stream is updated, as described in QUIC transport draft-23 section 3. Most of this can be inferred from several types of frames going over the wire, but it's much easier to have explicit signals for these state changes.

Definition:

```
StreamType = "unidirectional" / "bidirectional"

TransportStreamStateUpdated = {
  stream_id: uint64

  ; mainly useful when opening the stream
  ? stream_type: StreamType

  ? old: StreamState
  new: StreamState

  ? stream_side: "sending" / "receiving"
}

StreamState =
  ; bidirectional stream states, draft-23 3.4.
  "idle" /
  "open" /
  "half_closed_local" /
  "half_closed_remote" /
  "closed" /

  ; sending-side stream states, draft-23 3.1.
  "ready" /
  "send" /
  "data_sent" /
  "reset_sent" /
  "reset_received" /

  ; receive-side stream states, draft-23 3.2.
  "receive" /
  "size_known" /
  "data_read" /
  "reset_read" /

  ; both-side states
  "data_received" /

  ; qlog-defined:
  ; memory actually freed
  "destroyed"
```

Figure 21: TransportStreamStateUpdated definition

Note: QUIC implementations SHOULD mainly log the simplified bidirectional (HTTP/2-alike) stream states (e.g., idle, open, closed) instead of the more finegrained stream states (e.g., data_sent, reset_received). These latter ones are mainly for more in-depth debugging. Tools SHOULD be able to deal with both types equally.

3.3.14. frames_processed

Importance: Extra

This event's main goal is to prevent a large proliferation of specific purpose events (e.g., packets_acknowledged, flow_control_updated, stream_data_received). We want to give implementations the opportunity to (selectively) log this type of signal without having to log packet-level details (e.g., in packet_received). Since for almost all cases, the effects of applying a frame to the internal state of an implementation can be inferred from that frame's contents, we aggregate these events in this single "frames_processed" event.

Note: This event can be used to signal internal state change not resulting directly from the actual "parsing" of a frame (e.g., the frame could have been parsed, data put into a buffer, then later processed, then logged with this event).

Note: Implementations logging "packet_received" and which include all of the packet's constituent frames therein, are not expected to emit this "frames_processed" event. Rather, implementations not wishing to log full packets or that wish to explicitly convey extra information about when frames are processed (if not directly tied to their reception) can use this event.

Note: for some events, this approach will lose some information (e.g., for which encryption level are packets being acknowledged?). If this information is important, please use the packet_received event instead.

Note: in some implementations, it can be difficult to log frames directly, even when using packet_sent and packet_received events. For these cases, this event also contains the direct packet_number field, which can be used to more explicitly link this event to the packet_sent/received events.

Definition:

```
TransportFramesProcessed = {  
    ; see appendix for the QuicFrame definitions  
    frames: [* QuicFrame]  
  
    ? packet_number: uint64  
}
```

Figure 22: TransportFramesProcessed definition

3.3.15. data_moved

Importance: Base

Used to indicate when data moves between the different layers (for example passing from the application protocol (e.g., HTTP) to QUIC stream buffers and vice versa) or between the application protocol (e.g., HTTP) and the actual user application on top (for example a browser engine). This helps make clear the flow of data, how long data remains in various buffers and the overheads introduced by individual layers.

For example, this helps make clear whether received data on a QUIC stream is moved to the application protocol immediately (for example per received packet) or in larger batches (for example, all QUIC packets are processed first and afterwards the application layer reads from the streams with newly available data). This in turn can help identify bottlenecks or scheduling problems.

Definition:

```
TransportDataMoved = {  
    ? stream_id: uint64  
    ? offset: uint64  
  
    ; byte length of the moved data  
    ? length: uint64  
  
    ? from: "user" / "application" / "transport" / "network" / text  
    ? to: "user" / "application" / "transport" / "network" / text  
  
    ; raw bytes that were transferred  
    ? data: hexstring  
}
```

Figure 23: TransportDataMoved definition

Note: we do not for example use a "direction" field (with values "up" and "down") to specify the data flow. This is because in some optimized implementations, data might skip some individual layers. Additionally, using explicit "from" and "to" fields is more flexible and allows the definition of other conceptual "layers" (for example to indicate data from QUIC CRYPTO frames being passed to a TLS library ("security") or from HTTP/3 to QPACK ("qpack")).

Note: this event type is part of the "transport" category, but really spans all the different layers. This means we have a few leaky abstractions here (for example, the stream_id or stream offset might not be available at some logging points, or the raw data might not be in a byte-array form). In these situations, implementers can decide to define new, in-context fields to aid in manual debugging.

3.4. recovery

Note: most of the events in this category are kept generic to support different recovery approaches and various congestion control algorithms. Tool creators SHOULD make an effort to support and visualize even unknown data in these events (e.g., plot unknown congestion states by name on a timeline visualization).

3.4.1. parameters_set

Importance: Base

This event groups initial parameters from both loss detection and congestion control into a single event. All these settings are typically set once and never change. Implementation that do, for some reason, change these parameters during execution, MAY emit the parameters_set event twice.

Definition:


```
RecoveryParametersSet = {  
    ; Loss detection, see recovery draft-23, Appendix A.2  
    ; in amount of packets  
    ? reordering_threshold: uint16  
  
    ; as RTT multiplier  
    ? time_threshold: float32  
  
    ; in ms  
    timer_granularity: uint16  
  
    ; in ms  
    ? initial_rtt: float32  
  
    ; congestion control, Appendix B.1.  
    ; in bytes. Note: this could be updated after pmtud  
    ? max_datagram_size: uint32  
  
    ; in bytes  
    ? initial_congestion_window: uint64  
  
    ; Note: this could change when max_datagram_size changes  
    ; in bytes  
    ? minimum_congestion_window: uint32  
    ? loss_reduction_factor: float32  
  
    ; as PTO multiplier  
    ? persistent_congestion_threshold: uint16  
}
```

Figure 24: RecoveryParametersSet definition

Additionally, this event can contain any number of unspecified fields to support different recovery approaches.

3.4.2. metrics_updated

Importance: Core

This event is emitted when one or more of the observable recovery metrics changes value. This event SHOULD group all possible metric updates that happen at or around the same time in a single event (e.g., if min_rtt and smoothed_rtt change at the same time, they should be bundled in a single metrics_updated entry, rather than split out into two). Consequently, a metrics_updated event is only guaranteed to contain at least one of the listed metrics.

Definition:

```

RecoveryMetricsUpdated = {
    ; Loss detection, see recovery draft-23, Appendix A.3
    ; all following rtt fields are expressed in ms
    ? min_rtt: float32
    ? smoothed_rtt: float32
    ? latest_rtt: float32
    ? rtt_variance: float32

    ? pto_count: uint16

    ; Congestion control, Appendix B.2.
    ; in bytes
    ? congestion_window: uint64
    ? bytes_in_flight: uint64

    ; in bytes
    ? ssthresh: uint64

    ; qlog defined
    ; sum of all packet number spaces
    ? packets_in_flight: uint64

    ; in bits per second
    ? pacing_rate: uint64
}

```

Figure 25: RecoveryMetricsUpdated definition

Note: to make logging easier, implementations MAY log values even if they are the same as previously reported values (e.g., two subsequent RecoveryMetricsUpdated entries can both report the exact same value for min_rtt). However, applications SHOULD try to log only actual updates to values.

Additionally, this event can contain any number of unspecified fields to support different recovery approaches.

3.4.3. congestion_state_updated

Importance: Base

This event signifies when the congestion controller enters a significant new state and changes its behaviour. This event's definition is kept generic to support different Congestion Control algorithms. For example, for the algorithm defined in the Recovery draft ("enhanced" New Reno), the following states are defined:

- * slow_start

- * congestion_avoidance
- * application_limited
- * recovery

Definition:

```
RecoveryCongestionStateUpdated = {  
  ? old: text  
  new: text  
  
  ? trigger:  
    "persistent_congestion" /  
    "ECN"  
}
```

Figure 26: RecoveryCongestionStateUpdated definition

The "trigger" field SHOULD be logged if there are multiple ways in which a state change can occur but MAY be omitted if a given state can only be due to a single event occurring (e.g., slow start is exited only when ssthresh is exceeded).

3.4.4. loss_timer_updated

Importance: Extra

This event is emitted when a recovery loss timer changes state. The three main event types are:

- * set: the timer is set with a delta timeout for when it will trigger next
- * expired: when the timer effectively expires after the delta timeout
- * cancelled: when a timer is cancelled (e.g., all outstanding packets are acknowledged, start idle period)

Note: to indicate an active timer's timeout update, a new "set" event is used.

Definition:

```

RecoveryLossTimerUpdated = {
  ; called "mode" in draft-23 A.9.
  ? timer_type: "ack" / "pto"
  ? packet_number_space: PacketNumberSpace

  event_type: "set" / "expired" / "cancelled"

  ; if event_type === "set": delta time is in ms from
  ; this event's timestamp until when the timer will trigger
  ? delta: float32
}

```

Figure 27: RecoveryLossTimerUpdated definition

TODO: how about CC algo's that use multiple timers? How generic do these events need to be? Just support QUIC-style recovery from the spec or broader?

TODO: read up on the loss detection logic in draft-27 onward and see if this suffices

3.4.5. packet_lost

Importance: Core

This event is emitted when a packet is deemed lost by loss detection.

Definition:

```

RecoveryPacketLost = {
  ; should include at least the packet_type and packet_number
  ? header: PacketHeader

  ; not all implementations will keep track of full
  ; packets, so these are optional
  ; see appendix for the QuicFrame definitions
  ? frames: [* QuicFrame]

  ? trigger:
    "reordering_threshold" /
    "time_threshold" /
    ; draft-23 section 5.3.1, MAY
    "pto_expired"
}

```

Figure 28: RecoveryPacketLost definition

For this event, the "trigger" field SHOULD be set (for example to one of the values below), as this helps tremendously in debugging.

3.4.6. marked_for_retransmit

Importance: Extra

This event indicates which data was marked for retransmit upon detecting a packet loss (see packet_lost). Similar to our reasoning for the "frames_processed" event, in order to keep the amount of different events low, we group this signal for all types of retransmittable data in a single event based on existing QUIC frame definitions.

Implementations retransmitting full packets or frames directly can just log the constituent frames of the lost packet here (or do away with this event and use the contents of the packet_lost event instead). Conversely, implementations that have more complex logic (e.g., marking ranges in a stream's data buffer as in-flight), or that do not track sent frames in full (e.g., only stream offset + length), can translate their internal behaviour into the appropriate frame instance here even if that frame was never or will never be put on the wire.

Note: much of this data can be inferred if implementations log packet_sent events (e.g., looking at overlapping stream data offsets and length, one can determine when data was retransmitted).

Definition:

```
RecoveryMarkedForRetransmit = {  
    ; see appendix for the QuicFrame definitions  
    frames: [+ QuicFrame]  
}
```

Figure 29: RecoveryMarkedForRetransmit definition

4. Security Considerations

TBD

5. IANA Considerations

TBD

6. References

6.1. Normative References

- [CDDL] Birkholz, H., Vigano, C., and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures", RFC 8610, DOI 10.17487/RFC8610, June 2019, <<https://www.rfc-editor.org/rfc/rfc8610>>.
- [QLOG-H3] Marx, R., Ed., Niccolini, L., Ed., and M. Seemann, Ed., "HTTP/3 and QPACK event definitions for qlog", Work in Progress, Internet-Draft, draft-ietf-quic-qlog-h3-events-01, <<https://datatracker.ietf.org/doc/html/draft-ietf-quic-qlog-h3-events-01>>.
- [QLOG-MAIN] Marx, R., Ed., Niccolini, L., Ed., and M. Seemann, Ed., "Main logging schema for qlog", Work in Progress, Internet-Draft, draft-ietf-quic-qlog-main-schema-03, <<https://datatracker.ietf.org/doc/html/draft-ietf-quic-qlog-main-schema-03>>.
- [QUIC-RECOVERY] Iyengar, J., Ed. and I. Swett, Ed., "QUIC Loss Detection and Congestion Control", RFC 9002, DOI 10.17487/RFC9002, May 2021, <<https://www.rfc-editor.org/rfc/rfc9002>>.
- [QUIC-TLS] Thomson, M., Ed. and S. Turner, Ed., "Using TLS to Secure QUIC", RFC 9001, DOI 10.17487/RFC9001, May 2021, <<https://www.rfc-editor.org/rfc/rfc9001>>.
- [QUIC-TRANSPORT] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <<https://www.rfc-editor.org/rfc/rfc9000>>.

6.2. Informative References

- [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/rfc/rfc2119>>.

Appendix A. QUIC data field definitions

A.1. ProtocolEventBody extension

We extend the \$ProtocolEventBody extension point defined in [QLOG-MAIN] with the QUIC protocol events defined in this document.

```

QuicEvents = ConnectivityServerListening /
               ConnectivityConnectionStarted /
               ConnectivityConnectionClosed /
               ConnectivityConnectionIDUpdated /
               ConnectivitySpinBitUpdated /
               ConnectivityConnectionStateUpdated /
               SecurityKeyUpdated / SecurityKeyRetired /
               TransportVersionInformation / TransportALPNInformation /
               TransportParametersSet / TransportParametersRestored /
               TransportPacketSent / TransportPacketReceived /
               TransportPacketDropped / TransportPacketBuffered /
               TransportPacketsAacked / TransportDatagramsSent /
               TransportDatagramsReceived / TransportDatagramDropped /
               TransportStreamStateUpdated / TransportFramesProcessed /
               TransportDataMoved /
               RecoveryParametersSet / RecoveryMetricsUpdated /
               RecoveryCongestionStateUpdated /
               RecoveryLossTimerUpdated /
               RecoveryPacketLost

```

```
$ProtocolEventBody /= QuicEvents
```

A.2. QuicVersion

```
QuicVersion = hexstring
```

Figure 30: QuicVersion definition

A.3. ConnectionID

```
ConnectionID = hexstring
```

Figure 31: ConnectionID definition

A.4. Owner

```
Owner = "local" / "remote"
```

Figure 32: Owner definition

A.5. IPAddress and IPVersion

```

; an IPAddress can either be a "human readable" form
; (e.g., "127.0.0.1" for v4 or
; "2001:0db8:85a3:0000:0000:8a2e:0370:7334" for v6) or
; use a raw byte-form (as the string forms can be ambiguous)
IPAddress = text / hexstring

```

Figure 33: IPAddress definition

```
IPVersion = "v4" / "v6"
```

Figure 34: IPVersion definition

A.6. PacketType

```
PacketType = "initial" / "handshake" / "0RTT" / "1RTT" / "retry" /  
  "version_negotiation" / "stateless_reset" / "unknown"
```

Figure 35: PacketType definition

A.7. PacketNumberSpace

```
PacketNumberSpace = "initial" / "handshake" / "application_data"
```

Figure 36: PacketNumberSpace definition

A.8. PacketHeader

```
PacketHeader = {  
  packet_type: PacketType  
  packet_number: uint64  
  
  ; the bit flags of the packet headers (spin bit, key update bit,  
  ; etc. up to and including the packet number length bits  
  ; if present  
  ? flags: uint8  
  
  ; only if packet_type === "initial"  
  ? token: Token  
  
  ; only if packet_type === "initial" || "handshake" || "0RTT"  
  ; Signifies length of the packet_number plus the payload  
  ? length: uint16  
  
  ; only if present in the header  
  ; if correctly using transport:connection_id_updated events,  
  ; dcid can be skipped for 1RTT packets  
  ? version: QuicVersion  
  ? scil: uint8  
  ? dcil: uint8  
  ? scid: ConnectionID  
  ? dcid: ConnectionID  
}
```

Figure 37: PacketHeader definition

A.9. Token

```
Token = {  
  ? type: "retry" / "resumption" / "stateless_reset"  
  
  ; byte length of the token  
  ? length: uint32  
  
  ; raw byte value of the token  
  ? data: hexstring  
  
  ; decoded fields included in the token  
  ; (typically: peer's IP address, creation time)  
  ? details: {  
    * text => any  
  }  
}
```

Figure 38: Token definition

The token carried in an Initial packet can either be a retry token from a Retry packet, a stateless reset token from a Stateless Reset packet or one originally provided by the server in a NEW_TOKEN frame used when resuming a connection (e.g., for address validation purposes). Retry and resumption tokens typically contain encoded metadata to check the token's validity when it is used, but this metadata and its format is implementation specific. For that, this field includes a general-purpose "details" field.

A.10. KeyType

```
KeyType =  
  "server_initial_secret" / "client_initial_secret" /  
  "server_handshake_secret" / "client_handshake_secret" /  
  "server_0rtt_secret" / "client_0rtt_secret" /  
  "server_1rtt_secret" / "client_1rtt_secret"
```

Figure 39: KeyType definition

A.11. QUIC Frames

```

QuicFrame =
  PaddingFrame / PingFrame / AckFrame / ResetStreamFrame /
  StopSendingFrame / CryptoFrame / NewTokenFrame / StreamFrame /
  MaxDataFrame / MaxStreamDataFrame / MaxStreamsFrame /
  DataBlockedFrame / StreamDataBlockedFrame / StreamsBlockedFrame /
  NewConnectionIDFrame / RetireConnectionIDFrame /
  PathChallengeFrame / PathResponseFrame / ConnectionCloseFrame /
  HandshakeDoneFrame / UnknownFrame

```

Figure 40: QuicFrame definition

A.11.1. PaddingFrame

In QUIC, PADDING frames are simply identified as a single byte of value 0. As such, each padding byte could be theoretically interpreted and logged as an individual PaddingFrame.

However, as this leads to heavy logging overhead, implementations SHOULD instead emit just a single PaddingFrame and set the `payload_length` property to the amount of PADDING bytes/frames included in the packet.

```

PaddingFrame = {
  frame_type: "padding"

  ; total frame length, including frame header
  ? length: uint32
  payload_length: uint32
}

```

Figure 41: PaddingFrame definition

A.11.2. PingFrame

```

PingFrame = {
  frame_type: "ping"

  ; total frame length, including frame header
  ? length: uint32
  ? payload_length: uint32
}

```

Figure 42: PingFrame definition

A.11.3. AckFrame

```
; either a single number (e.g., [1]) or two numbers (e.g., [1,2]).
; For two numbers:
; the first number is "from": lowest packet number in interval
; the second number is "to": up to and including the highest
; packet number in the interval
AckRange = [1*2 uint64]

AckFrame = {
    frame_type: "ack"

    ; in ms
    ? ack_delay: float32

    ; e.g., looks like [[1,2],[4,5], [7], [10,22]] serialized
    ? acked_ranges: [+ AckRange]

    ; ECN (explicit congestion notification) related fields
    ; (not always present)
    ? ect1: uint64
    ? ect0:uint64
    ? ce: uint64

    ; total frame length, including frame header
    ? length: uint32
    ? payload_length: uint32
}
```

Figure 43: AckFrame definition

Note: the packet ranges in AckFrame.acked_ranges do not necessarily have to be ordered (e.g., [[5,9],[1,4]] is a valid value).

Note: the two numbers in the packet range can be the same (e.g., [120,120] means that packet with number 120 was ACKed). However, in that case, implementers SHOULD log [120] instead and tools MUST be able to deal with both notations.

A.11.4. ResetStreamFrame

```
ResetStreamFrame = {  
    frame_type: "reset_stream"  
  
    stream_id: uint64  
    error_code: $ApplicationError / uint32  
  
    ; in bytes  
    final_size: uint64  
  
    ; total frame length, including frame header  
    ? length: uint32  
    ? payload_length: uint32  
}
```

Figure 44: ResetStreamFrame definition

A.11.5. StopSendingFrame

```
StopSendingFrame = {  
    frame_type: "stop_sending"  
  
    stream_id: uint64  
    error_code: $ApplicationError / uint32  
  
    ; total frame length, including frame header  
    ? length: uint32  
    ? payload_length: uint32  
}
```

Figure 45: StopSendingFrame definition

A.11.6. CryptoFrame

```
CryptoFrame = {  
    frame_type: "crypto"  
  
    offset: uint64  
    length: uint64  
  
    ? payload_length: uint32  
}
```

Figure 46: CryptoFrame definition

A.11.7. NewTokenFrame

```
NewTokenFrame = {  
    frame_type: "new_token"  
  
    token: Token  
}
```

Figure 47: NewTokenFrame definition

A.11.8. StreamFrame

```
StreamFrame = {  
    frame_type: "stream"  
  
    stream_id: uint64  
  
    ; These two MUST always be set  
    ; If not present in the Frame type, log their default values  
    offset: uint64  
    length: uint64  
  
    ; this MAY be set any time,  
    ; but MUST only be set if the value is true  
    ; if absent, the value MUST be assumed to be false  
    ? fin: bool .default false  
  
    ? raw: hexstring  
}
```

Figure 48: StreamFrame definition

A.11.9. MaxDataFrame

```
MaxDataFrame = {  
    frame_type: "max_data"  
  
    maximum: uint64  
}
```

Figure 49: MaxDataFrame definition

A.11.10. MaxStreamDataFrame

```
MaxStreamDataFrame = {  
    frame_type: "max_stream_data"  
  
    stream_id: uint64  
    maximum: uint64  
}
```

Figure 50: MaxStreamDataFrame definition

A.11.11. MaxStreamsFrame

```
MaxStreamsFrame = {  
  frame_type: "max_streams"  
  
  stream_type: StreamType  
  maximum: uint64  
}
```

Figure 51: MaxStreamsFrame definition

A.11.12. DataBlockedFrame

```
DataBlockedFrame = {  
  frame_type: "data_blocked"  
  
  limit: uint64  
}
```

Figure 52: DataBlockedFrame definition

A.11.13. StreamDataBlockedFrame

```
StreamDataBlockedFrame = {  
  frame_type: "stream_data_blocked"  
  
  stream_id: uint64  
  limit: uint64  
}
```

Figure 53: StreamDataBlockedFrame definition

A.11.14. StreamsBlockedFrame

```
StreamsBlockedFrame = {  
  frame_type: "streams_blocked"  
  
  stream_type: StreamType  
  limit: uint64  
}
```

Figure 54: StreamsBlockedFrame definition

A.11.15. NewConnectionIDFrame

```
NewConnectionIDFrame = {  
  frame_type: "new_connection_id"  
  
  sequence_number: uint32  
  retire_prior_to: uint32  
  
  ; mainly used if e.g., for privacy reasons the full  
  ; connection_id cannot be logged  
  ? connection_id_length: uint8  
  connection_id: ConnectionID  
  
  ? stateless_reset_token: Token  
}
```

Figure 55: NewConnectionIDFrame definition

A.11.16. RetireConnectionIDFrame

```
RetireConnectionIDFrame = {  
  frame_type: "retire_connection_id"  
  
  sequence_number: uint32  
}
```

Figure 56: RetireConnectionIDFrame definition

A.11.17. PathChallengeFrame

```
PathChallengeFrame = {  
  frame_type: "path_challenge"  
  
  ; always 64-bit  
  ? data: hexstring  
}
```

Figure 57: PathChallengeFrame definition

A.11.18. PathResponseFrame

```
PathResponseFrame = {  
  frame_type: "path_response"  
  
  ; always 64-bit  
  ? data: hexstring  
}
```

Figure 58: PathResponseFrame definition

A.11.19. ConnectionCloseFrame

raw_error_code is the actual, numerical code. This is useful because some error types are spread out over a range of codes (e.g., QUIC's crypto_error).

ErrorSpace = "transport" / "application"

```
ConnectionCloseFrame = {
  frame_type: "connection_close"

  ? error_space: ErrorSpace
  ? error_code: TransportError / $ApplicationError / uint32
  ? raw_error_code: uint32
  ? reason: text

  ; For known frame types, the appropriate "frame_type" string
  ; For unknown frame types, the hex encoded identifier value
  ? trigger_frame_type: uint64 / text
}
```

Figure 59: ConnectionCloseFrame definition

A.11.20. HandshakeDoneFrame

```
HandshakeDoneFrame = {
  frame_type: "handshake_done";
}
```

Figure 60: HandshakeDoneFrame definition

A.11.21. UnknownFrame

```
UnknownFrame = {
  frame_type: "unknown"
  raw_frame_type: uint64

  ? raw_length: uint32
  ? raw: hexstring
}
```

Figure 61: UnknownFrame definition

A.11.22. TransportError


```

TransportError = "no_error" / "internal_error" /
  "connection_refused" / "flow_control_error" /
  "stream_limit_error" / "stream_state_error" /
  "final_size_error" / "frame_encoding_error" /
  "transport_parameter_error" / "connection_id_limit_error" /
  "protocol_violation" / "invalid_token" / "application_error" /
  "crypto_buffer_exceeded"

```

Figure 62: TransportError definition

A.11.23. ApplicationError

By definition, an application error is defined by the application-level protocol running on top of QUIC (e.g., HTTP/3).

As such, we cannot define it here directly. Though we provide an extension point through the use of the CDDL "socket" mechanism.

Application-level qlog definitions that wish to define new ApplicationError strings MUST do so by extending the \$ApplicationError socket as such:

```
$ApplicationError /= "new_error_name" / "another_new_error_name"
```

A.11.24. CryptoError

These errors are defined in the TLS document as "A TLS alert is turned into a QUIC connection error by converting the one-byte alert description into a QUIC error code. The alert description is added to 0x100 to produce a QUIC error code from the range reserved for CRYPTO_ERROR."

This approach maps badly to a pre-defined enum. As such, we define the crypto_error string as having a dynamic component here, which should include the hex-encoded and zero-padded value of the TLS alert description.

```

; all strings from "crypto_error_0x100" to "crypto_error_0x199"
CryptoError = text .regexp "crypto_error_0x1[0-9][0-9]"

```

Figure 63: CryptoError definition

Appendix B. Change Log

B.1. Since draft-ietf-qlog-quic-events-00:

- * Change the data definition language from TypeScript to CDDL (#143)

B.2. Since draft-marx-qlog-event-definitions-quic-h3-02:

- * These changes were done in preparation of the adoption of the drafts by the QUIC working group (#137)
- * Split QUIC and HTTP/3 events into two separate documents
- * Moved RawInfo, Importance, Generic events and Simulation events to the main schema document.
- * Changed to/from value options of the data_moved event

B.3. Since draft-marx-qlog-event-definitions-quic-h3-01:

Major changes:

- * Moved data_moved from http to transport. Also made the "from" and "to" fields flexible strings instead of an enum (#111,#65)
- * Moved packet_type fields to PacketHeader. Moved packet_size field out of PacketHeader to RawInfo:length (#40)
- * Made events that need to log packet_type and packet_number use a header field instead of logging these fields individually
- * Added support for logging retry, stateless reset and initial tokens (#94,#86,#117)
- * Moved separate general event categories into a single category "generic" (#47)
- * Added "transport:connection_closed" event (#43,#85,#78,#49)
- * Added version_information and alpn_information events (#85,#75,#28)
- * Added parameters_restored events to help clarify 0-RTT behaviour (#88)

Smaller changes:

- * Merged loss_timer events into one loss_timer_updated event
- * Field data types are now strongly defined (#10,#39,#36,#115)
- * Renamed qpack instruction_received and instruction_sent to instruction_created and instruction_parsed (#114)

- * Updated `qpack:dynamic_table_updated.update_type`. It now has the value "inserted" instead of "added" (#113)
- * Updated `qpack:dynamic_table_updated`. It now has an "owner" field to differentiate encoder vs decoder state (#112)
- * Removed `push_allowed` from `http:parameters_set` (#110)
- * Removed explicit trigger field indications from events, since this was moved to be a generic property of the "data" field (#80)
- * Updated `transport:connection_id_updated` to be more in line with other similar events. Also dropped importance from Core to Base (#45)
- * Added length property to `PaddingFrame` (#34)
- * Added `packet_number` field to `transport:frames_processed` (#74)
- * Added a way to generically log packet header flags (first 8 bits) to `PacketHeader`
- * Added additional guidance on which events to log in which situations (#53)
- * Added "simulation:scenario" event to help indicate simulation details
- * Added "packets_acked" event (#107)
- * Added "datagram_ids" to the `datagram_X` and `packet_X` events to allow tracking of coalesced QUIC packets (#91)
- * Extended `connection_state_updated` with more fine-grained states (#49)

B.4. Since draft-marx-qlog-event-definitions-quic-h3-00:

- * Event and category names are now all lowercase
- * Added many new events and their definitions
- * "type" fields have been made more specific (especially important for `PacketType` fields, which are now called `packet_type` instead of `type`)
- * Events are given an importance indicator (issue #22)

- * Event names are more consistent and use past tense (issue #21)
- * Triggers have been redefined as properties of the "data" field and updated for most events (issue #23)

Appendix C. Design Variations

TBD

Appendix D. Acknowledgements

Much of the initial work by Robin Marx was done at Hasselt University.

Thanks to Marten Seemann, Jana Iyengar, Brian Trammell, Dmitri Tikhonov, Stephen Petrides, Jari Arkko, Marcus Ihlar, Victor Vasiliev, Mirja Kuehlewind, Jeremy Laine, Kazu Yamamoto, Christian Huitema, and Lucas Pardue for their feedback and suggestions.

Authors' Addresses

Robin Marx
KU Leuven
Email: robin.marx@kuleuven.be

Luca Niccolini (editor)
Facebook
Email: lniccolini@fb.com

Marten Seemann (editor)
Protocol Labs
Email: marten@protocol.ai

QUIC
Internet-Draft
Intended status: Standards Track
Expires: 8 October 2022

D. Schinazi
Google LLC
E. Rescorla
Mozilla
6 April 2022

Compatible Version Negotiation for QUIC
draft-ietf-quic-version-negotiation-07

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 chose is unacceptable. This document describes a version negotiation mechanism that allows a client and server to select a mutually supported version. Optionally, if the client's chosen version and the negotiated version share a compatible first flight format, the negotiation can take place without incurring an extra round trip.

About This Document

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

The latest revision of this draft can be found at <https://quicwg.github.io/version-negotiation/draft-ietf-quic-version-negotiation.html>. Status information for this document may be found at <https://datatracker.ietf.org/doc/draft-ietf-quic-version-negotiation/>.

Discussion of this document takes place on the QUIC Working Group mailing list (<mailto:quic@ietf.org>), which is archived at <https://mailarchive.ietf.org/arch/browse/quic/>.

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 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 October 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction	3
1.1. Conventions and Definitions	3
2. Version Negotiation Mechanism	3
2.1. Incompatible Version Negotiation	4
2.2. Compatible Versions	5
2.3. Compatible Version Negotiation	6
2.4. Connections and Version Negotiation	7
2.5. Client Choice of Original Version	8
3. Version Information	8
4. Version Downgrade Prevention	9
5. Server Deployments of QUIC	10
6. Application Layer Protocol Considerations	12
7. Considerations for Future Versions	12
7.1. Interaction with Retry	13
7.2. Interaction with TLS resumption	13
7.3. Interaction with 0-RTT	13
8. Special Handling for QUIC Version 1	14
9. Security Considerations	14
10. IANA Considerations	14
10.1. QUIC Transport Parameter	14
10.2. QUIC Transport Error Code	15
11. Normative References	15
Acknowledgments	16
Authors' Addresses	16

1. Introduction

The version-invariant properties of QUIC [INV] define a Version Negotiation packet but do not specify how an endpoint reacts when it receives one. QUIC version 1 [QUIC] allows the server to use a Version Negotiation packet to indicate that the version the client chose 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 Version Negotiation packet can be part of a mechanism to allow two QUIC implementations to negotiate between two totally disjoint versions of QUIC. This document specifies version negotiation using Version Negotiation packets, which adds an extra round trip to connection establishment if needed.

It is beneficial to avoid additional round trips whenever possible, especially given that most incremental versions are broadly similar to the the previous version. This specification also defines a simple version negotiation mechanism which leverages similarities between versions and can negotiate between the set of "compatible" versions without additional round trips.

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.

In this document, the Maximum Segment Lifetime (MSL) represents the time a QUIC packet can exist in the network. Implementations can make this configurable, and a RECOMMENDED value is one minute.

2. 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 choosing an initial version and sending a first flight of QUIC packets with a long header to the server [INV]. The client's first flight includes Version Information (see Section 3) which will be used to optionally enable compatible version negotiation (see Section 2.3), and to prevent version

downgrade attacks (see Section 4). We'll refer to the version of the very first packets the client sends as the "original version" and the version of the first packets the client sends in a given QUIC connection as the "client's chosen version".

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 2.1, which causes the client to initiate a new connection with a different version. For instance, if the client initiates a connection with version A and the server starts incompatible version negotiation and the client then initiates a new connection with version B, we say that the first connection's client chosen version is A, the second connection's client chosen version is B, and the original version for the entire sequence is A.

If the server can parse the first flight, it can either establish the connection using the client's chosen version, or it MAY select any other compatible version, as described in Section 2.3.

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.

2.1. Incompatible Version Negotiation

The server starts incompatible version negotiation by sending a Version Negotiation packet. This packet SHALL include each entry from the server's set of Offered Versions (see Section 5) in a Supported Version field. The server MAY add reserved versions (as defined in Section 6.3 of [QUIC]) in Supported Version fields.

Clients will ignore a Version Negotiation packet if it contains the original version attempted by the client. The client also ignores a Version Negotiation packet that contains incorrect connection ID fields; see Section 6 of [INV].

Upon receiving the Version Negotiation 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 version information (see Section 3) required to ensure that version negotiation was genuine, i.e. that no attacker injected packets in order to influence the version negotiation process, see Section 4.

2.2. 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. Note that the conversion of the first flight can be lossy: some data such as QUIC version 1 0-RTT packets could be ignored during conversion and retransmitted later.

Version compatibility is not symmetric: 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: if version A includes the concept of streams and STREAM frames, and version B includes the concepts of streams and tubes along with STREAM and TUBE frames, then A would be compatible with B but B would not be compatible with 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.

2.3. Compatible Version Negotiation

When the server can parse the client's first flight using the client's chosen version, it can extract the client's Version Information structure (see Section 3). This contains the list of versions that the client knows its first flight is compatible with.

In order to perform compatible version negotiation, the server **MUST** select one of these versions that (1) it supports and (2) it knows the client's chosen version to be compatible with. Once the server has selected a version, termed the "negotiated version", it then attempts to convert the client's first flight into that version, and replies to the client as if it had received the converted first flight.

If those formats are identical, as in cases where the negotiated version is the same as the client's chosen version, then this will be the identity transform. If the first flight is correctly formatted, then this conversion process cannot fail by definition of the first flight being compatible; if the server is unable to convert the first flight, it **MUST** abort the handshake.

Clients can determine the server's negotiated version by examining the QUIC long header Version field. It is possible for the server to initially send packets with the client's chosen version before switching to the negotiated version (for example, this can happen when the client's Version Information structure spans multiple packets; in that case the server might acknowledge the first packet in the client's chosen version and later switch to a different negotiated version).

Note that, after the first flight is converted to the negotiated version, the handshake completes in the negotiated version. The entire handshake (including the converted first flight) needs to conform to the rules of the negotiated version. For instance, if the negotiated version requires that the 5-tuple remain stable for the entire handshake (as QUIC version 1 does), then this applies to the entire handshake, including the first flight.

Note also that the client can disable compatible version negotiation by only including the Chosen Version in the Other Versions field of the Version Information transport parameter.

If the server does not find a compatible version (including the client's chosen version), it will perform incompatible version negotiation instead, see Section 2.1.

Note that it is possible to have incompatible version negotiation followed by compatible version negotiation. For instance, if version A is compatible with B and C is compatible with D, the following scenario could occur:

Client	Server
Chosen = A, Other Versions = (A, B) ----->	
<-----	Version Negotiation = (D, C)
Chosen = C, Other Versions = (C, D) ----->	
<-----	Negotiated = D

Figure 1: Combined Negotiation Example

In this example, the client selected C from the server's Version Negotiation packet, but the server preferred D and then selected it from the client's offer.

2.4. 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 2.3) is performed as part of a single QUIC connection; that is, the packets with the client's chosen version are part of the same connection as the packets with the negotiated version.

In comparison, the incompatible version negotiation mechanism, which leverages QUIC Version Negotiation packets (see Section 2.1) conceptually operates across two QUIC connections: the connection attempt prior to receiving the Version Negotiation packet is distinct from the connection with the incompatible version that follows.

Note that this separation across two connections is conceptual: it applies to normative requirements on QUIC connections, but does not require implementations to internally use two distinct connection objects.

2.5. Client Choice of Original Version

When the client picks its original version, it will try to avoid incompatible version negotiation to save a round trip. Therefore, the client SHOULD pick an original version to maximize the combined probability that both:

- * The server knows how to parse first flights from the original version.
- * The original version is compatible with the client's preferred version.

Without additional information, this could mean selecting the oldest version that the client supports.

3. Version Information

During the handshake, endpoints will exchange Version Information, which consists of a chosen version and a list of other versions. Any version of QUIC that supports this mechanism MUST provide a mechanism to exchange Version Information in both directions during the handshake, such that this data is authenticated.

In QUIC version 1, the Version Information is transmitted using a new transport parameter, `version_information`. The contents of Version Information are shown below (using the notation from the "Notational Conventions" section of [QUIC]):

```
Version Information {  
    Chosen Version (32),  
    Other Versions (32) ...,  
}
```

Figure 2: Version Information Format

The content of each field is described below:

Chosen Version: The version that the sender has chosen to use for this connection. In most cases, this field will be equal to the value of the Version field in the long header that carries this data.

The contents of the Other Versions field depends on whether it is sent by the client or by the server.

Client-Sent Other Versions: When sent by a client, the Other

Versions field lists all the versions that this first flight is compatible with, ordered by descending preference. Note that the version in the Chosen Version field MUST be included in this list to allow the client to communicate the chosen version's preference. Note that this preference is only advisory, servers MAY choose to use their own preference instead.

Server-Sent Other Versions: When sent by a server, the Other Versions field lists all the Fully-Deployed Versions of this server deployment, see Section 5. Note that the version in the Chosen Version field is not necessarily included in this list, because the server operator could be in the process of removing support for this version. For the same reason, the Other Versions field MAY be empty.

Clients and servers MAY both include versions following the pattern 0x?a?a?a in their Other Versions list. Those versions are reserved to exercise version negotiation (see the Versions section of [QUIC]), and will never be selected when choosing a version to use.

4. Version Downgrade Prevention

Clients MUST ignore any received Version Negotiation packets that contain the version that they initially attempted. A client that makes a connection attempt based on information received from a Version Negotiation packet MUST ignore any Version Negotiation packets it receives in response to that connection attempt.

Both endpoints MUST parse their peer's Version Information during the handshake. If parsing the Version Information failed (for example, if it is too short or if its length is not divisible by four), then the endpoint MUST close the connection; if the connection was using QUIC version 1, that connection closure MUST use a transport error of type `TRANSPORT_PARAMETER_ERROR`. If an endpoint receives a Chosen Version equal to zero, or any Other Version equal to zero, it MUST treat it as a parsing failure.

Every QUIC version that supports version negotiation MUST define a method for closing the connection with a version negotiation error. For QUIC version 1, version negotiation errors are signaled using a transport error of type `VERSION_NEGOTIATION_ERROR`; see Section 10.2.

If the Version Information was missing, the endpoints MAY complete the handshake. However, if a client has reacted to a Version Negotiation packet and the Version Information was missing, the client MUST close the connection with a version negotiation error.

If the client received and acted on a Version Negotiation packet, the client MUST validate the server's Other Versions field. The Other Versions field is validated by confirming that the client would have attempted the same version with knowledge of the versions the server supports. That is, the client would have selected the same version if it received a Version Negotiation packet that listed the versions in the server's Other Versions field, plus the negotiated version. If the client would have selected a different version, the client MUST close the connection with a version negotiation error. In particular, if the client reacted to a Version Negotiation packet and the server's Other Versions field is empty, the client MUST close the connection with a version negotiation error. These connection closures prevent an attacker from being able to use forged Version Negotiation packets to force a version downgrade.

This validation of Other Versions is not sufficient to prevent downgrade. Downgrade prevention also depends on the client ignoring Version Negotiation packets that contain the original version; see Section 2.1.

After the process of version negotiation in this document completes, the version in use for the connection is the version that the server sent in the Chosen Version field of its Version Information. That remains true even if other versions were used in the Version field of long headers at any point in the lifetime of the connection. In particular, since during compatible version negotiation the client is made aware of the negotiated version by the QUIC long header version (see Section 2.3), clients MUST validate that the server's Chosen Version is equal to the negotiated version; if they do not match, the client MUST close the connection with a version negotiation error. This prevents an attacker's ability to influence version negotiation by forging the Version long header field.

5. Server Deployments of QUIC

While this document mainly discusses a single QUIC server, it is common for deployments of QUIC servers to include a fleet of multiple server instances. We therefore define the following terms:

Acceptable Versions: This is the set of versions supported by a given server instance. More specifically, these are the versions that a given server instance will use if a client sends a first flight using them.

Offered Versions: This is the set of versions that a given server instance will send in a Version Negotiation packet if it receives a first flight from an unknown version. This set will most often be equal to the Acceptable Versions set, except during short transitions while versions are added or removed (see below).

Fully-Deployed Versions: This is the set of QUIC versions that is supported and negotiated by every single QUIC server instance in this deployment. If a deployment only contains a single server instance, then this set is equal to the Offered Versions set, except during short transitions while versions are added or removed (see below).

If a deployment contains multiple server instances, software updates may not happen at exactly the same time on all server instances. Because of this, a client might receive a Version Negotiation packet from a server instance that has already been updated and the client's resulting connection attempt might reach a different server instance which hasn't been updated yet.

However, even when there is only a single server instance, it is still possible to receive a stale Version Negotiation packet if the server performs its software update while the Version Negotiation packet is in flight.

This could cause the version downgrade prevention mechanism described in Section 4 to falsely detect a downgrade attack. To avoid that, server operators SHOULD perform a three-step process when they wish to add or remove support for a version:

When adding support for a new version:

- * The first step is to progressively add support for the new version to all server instances. This step updates the Acceptable Versions but not the Offered Versions nor the Fully-Deployed Versions. Once all server instances have been updated, operators wait for at least one MSL to allow any in-flight Version Negotiation packets to arrive.
- * Then, the second step is to progressively add the new version to Offered Versions on all server instances. Once complete, operators wait for at least another MSL.
- * Finally, the third step is to progressively add the new version to Fully-Deployed Versions on all server instances.

When removing support for a version:

- * The first step is to progressively remove the version from Fully-Deployed Versions on all server instances. Once it has been removed on all server instances, operators wait for at least one MSL to allow any in-flight Version Negotiation packets to arrive.

- * Then, the second step is to progressively remove the version from Offered Versions on all server instances. Once complete, operators wait for at least another MSL.
- * Finally, the third step is to progressively remove support for the version from all server instances. That step updates the Acceptable Versions.

Note that this opens connections to version downgrades (but only for partially-deployed versions) during the update window, since those could be due to clients communicating with both updated and non-updated server instances.

6. Application Layer Protocol Considerations

When a client creates a QUIC connection, its goal is to use an application layer protocol. Therefore, when considering which versions are compatible, clients will only consider versions that support one of the intended application layer protocols. If the client's first flight advertises multiple Application Layer Protocol Negotiation (ALPN) [ALPN] tokens and multiple compatible versions, it is possible for some application layer protocols to not be able to run over some of the offered compatible versions. It is the server's responsibility to only select an ALPN token that can run over the compatible QUIC version that it selects.

A given ALPN token MUST NOT be used with a new QUIC version different from the version for which the ALPN token was originally defined, unless all the following requirements are met:

- * The new QUIC version supports the transport features required by the application protocol.
- * The new QUIC version supports ALPN.
- * The version of QUIC for which the ALPN token was originally defined is compatible with the new QUIC version.

When incompatible version negotiation is in use, the second connection which is created in response to the received version negotiation packet MUST restart its application layer protocol negotiation process without taking into account the original version.

7. 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.

QUIC version 1 defines multiple features which are not documented in the QUIC invariants. Since at the time of writing QUIC version 1 is widely deployed, this section discusses considerations for future versions to help with compatibility with QUIC version 1.

7.1. 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. A server that sends a Retry packet can do so before parsing the client's first flight. A server that sends a Retry packet therefore might not have processed the client's Version Information before doing so.

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 in the original version first thereby validating the client's IP address before attempting compatible version negotiation. If both versions support authenticating Retry packets, the compatibility definition needs to define how to authenticate the Retry in the negotiated version handshake even though the Retry itself was sent using the client's chosen version.

7.2. Interaction with TLS resumption

QUIC version 1 uses TLS 1.3, which supports session resumption by sending session tickets in one connection that can be used in a later connection; see Section 2.2 of [TLS]. New versions that also use TLS 1.3 **SHOULD** mandate that their session tickets are tightly scoped to one version of QUIC; i.e., require that clients not use them across multiple version and that servers validate this client requirement.

7.3. 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. That document could specify that compatible version negotiation causes 0-RTT data to be rejected by the server.

8. Special Handling for QUIC Version 1

Because QUIC version 1 was the only IETF Standards Track version of QUIC published before this document, it is handled specially as follows: if a client is starting a QUIC version 1 connection in response to a received Version Negotiation packet, and the `version_information` transport parameter is missing from the server's transport parameters, then the client SHALL proceed as if the server's transport parameters contained a `version_information` transport parameter with a Chosen Version set to 0x00000001 and an Other Version list containing exactly one version set to 0x00000001. This allows version negotiation to work with servers that only support QUIC version 1. Note that implementations which wish to use version negotiation to negotiate versions other than QUIC version 1 will need to implement the version negotiation mechanism defined in this document.

9. Security Considerations

The security of this version negotiation mechanism relies on the authenticity of the 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.

10. IANA Considerations

10.1. QUIC Transport Parameter

This document registers a new value in the "QUIC Transport Parameters" registry maintained at <https://www.iana.org/assignments/quic>.

Value: 0xFF73DB
Parameter Name: `version_information`
Status: provisional
Specification: This document

When this document is approved, it will request permanent allocation of a codepoint in the 0-63 range to replace the provisional codepoint described above.

10.2. QUIC Transport Error Code

This document registers a new value in the "QUIC Transport Error Codes" registry maintained at <https://www.iana.org/assignments/quic>.

Value: 0x53F8
Code: VERSION_NEGOTIATION_ERROR
Description: Error negotiating version
Status: provisional
Specification: This document

When this document is approved, it will request permanent allocation of a codepoint in the 0-63 range to replace the provisional codepoint described above.

11. Normative References

- [ALPN] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <https://www.rfc-editor.org/rfc/rfc7301>.
- [INV] Thomson, M., "Version-Independent Properties of QUIC", RFC 8999, DOI 10.17487/RFC8999, May 2021, <https://www.rfc-editor.org/rfc/rfc8999>.
- [QUIC] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <https://www.rfc-editor.org/rfc/rfc9000>.
- [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/rfc/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/rfc/rfc8174>.
- [TLS] 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>.

Acknowledgments

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

Authors' Addresses

David Schinazi
Google LLC
1600 Amphitheatre Parkway
Mountain View, CA 94043
United States of America
Email: dschinazi.ietf@gmail.com

Eric Rescorla
Mozilla
Email: ekr@rtfm.com