

QUIC
Internet-Draft
Intended status: Standards Track
Expires: 10 January 2022

M. Duke
F5 Networks, Inc.
9 July 2021

QUIC Version 2
draft-duke-quic-v2-02

Abstract

This document specifies QUIC version 2, which is identical to QUIC version 1 except for some trivial details. Its purpose is to combat various ossification vectors and exercise the version negotiation framework. Over time, it may also serve as a vehicle for needed protocol design changes.

Discussion of this work is encouraged to happen on the QUIC IETF mailing list quic@ietf.org or on the GitHub repository which contains the draft: <https://github.com/martinduke/draft-duke-quic-v2>.

Discussion Venues

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

Discussion of this document takes place on the mailing list (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/martinduke/draft-duke-quic-v2>.

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 10 January 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	3
3. Changes from QUIC Version 1	3
4. Version Negotiation Considerations	4
5. Ossification Considerations	4
6. Applicability	5
7. Security Considerations	5
8. IANA Considerations	5
9. References	5
9.1. Normative References	5
9.2. Informative References	6
Appendix A. Changelog	6
A.1. since draft-duke-quic-v2-01	6
A.2. since draft-duke-quic-v2-00	6
Author's Address	7

1. Introduction

QUIC [RFC9000] has numerous extension points, including the version number that occupies the second through fifth octets of every long header (see [RFC8999]). If experimental versions are rare, and QUIC version 1 constitutes the vast majority of QUIC traffic, there is the potential for middleboxes to ossify on the version octets always being 0x00000001.

Furthermore, version 1 Initial packets are encrypted with keys derived from a universally known salt, which allow observers to inspect the contents of these packets, which include the TLS Client Hello and Server Hello messages. Again, middleboxes may ossify on the version 1 key derivation and packet formats.

Finally [QUIC-VN] provides two mechanisms for endpoints to negotiate the QUIC version to use. The "incompatible" version negotiation method can support switching from any initial QUIC version to any other version with full generality, at the cost of an additional round-trip at the start of the connection. "Compatible" version negotiation eliminates the round-trip penalty but levies some restrictions on how much the two versions can differ semantically.

QUIC version 2 is meant to mitigate ossification concerns and exercise the version negotiation mechanisms. The only change is a tweak to the inputs of some crypto derivation functions to enforce full key separation. Any endpoint that supports two versions needs to implement version negotiation to protect against downgrade attacks.

This document may, over time, also serve as a vehicle for other needed changes to QUIC version 1.

[I-D.duke-quic-version-aliasing] is a more robust, but much more complicated, proposal to address these ossification problems. By design, it requires incompatible version negotiation. QUICv2 enables exercise of compatible version negotiation mechanism.

2. 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 RFC 2119 [RFC2119].

3. Changes from QUIC Version 1

QUIC version 2 endpoints MUST implement the QUIC version 1 specification as described in [RFC9000], [RFC9001], and [RFC9002], with the following changes:

- * The version field of long headers is TBD. Note: Unless this document is published as an RFC, implementations should use the provisional value 0xff010001, which might change with each edition of this document.
- * The salt used to derive Initial keys in Sec 5.2 of [RFC9001] changes to

```
initial_salt = 0xa707c203a59b47184a1d62ca570406ea7ae3e5d3
```

- * The labels used in [RFC9001] to derive packet protection keys (Sec 5.1), header protection keys (Sec 5.4), Retry Integrity Tag keys (Sec 5.8), and key updates (Sec 6.1) change from "quic key" to

"quicv2 key", from "quic iv" to "quicv2 iv", from "quic hp" to "quicv2 hp", and from "quic ku" to "quicv2 ku," to meet the guidance for new versions in Section 9.6 of that document.

- * The key and nonce used for the Retry Integrity Tag (Sec 5.8 of [RFC9001]) change to:

```
secret = 0x3425c20cf88779df2ff71e8abfa78249891e763bbed2f13c048343d348c060e2
key = 0xba858dc7b43de5dbf87617ff4ab253db
nonce = 0x141b99c239b03e785d6a2e9f
```

4. Version Negotiation Considerations

QUIC version 2 endpoints SHOULD also support QUIC version 1. Any QUIC endpoint that supports multiple versions MUST fully implement [QUIC-VN] to prevent version downgrade attacks.

Note that version 2 meets that document's definition of a compatible version with version 1. Therefore, v2-capable servers MUST use compatible version negotiation unless they do not support version 1.

As version 1 support is more likely than version 2 support, a client SHOULD use QUIC version 1 for its original version unless it has out-of-band knowledge that the server supports version 2.

5. Ossification Considerations

QUIC version 2 provides protection against some forms of ossification. Devices that assume that all long headers will contain encode version 1, or that the version 1 Initial key derivation formula will remain version-invariant, will not correctly process version 2 packets.

However, many middleboxes such as firewalls focus on the first packet in a connection, which will often remain in the version 1 format due to the considerations above.

Clients interested in combating firewall ossification can initiate a connection using version 2 if they are either reasonably certain the server supports it, or are willing to suffer a round-trip penalty if they are incorrect.

6. Applicability

This version of QUIC provides no change from QUIC version 1 relating to the capabilities available to applications. Therefore, all Application Layer Protocol Negotiation (ALPN) ([RFC7301]) codepoints specified to operate over QUICv1 can also operate over this version of QUIC.

7. Security Considerations

QUIC version 2 introduces no changes to the security or privacy properties of QUIC version 1.

The mandatory version negotiation mechanism guards against downgrade attacks, but downgrades have no security implications, as the version properties are identical.

8. IANA Considerations

This document requests that IANA add the following entry to the QUIC version registry:

Value: TBD

Status: permanent

Specification: This Document

Change Controller: IETF

Contact: QUIC WG

9. References

9.1. Normative References

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

[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>>.

- [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/info/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/info/rfc9002>>.

9.2. Informative References

- [I-D.duke-quic-version-aliasing]
Duke, M., "QUIC Version Aliasing", Work in Progress, Internet-Draft, draft-duke-quic-version-aliasing-04, 30 October 2020, <<https://www.ietf.org/archive/id/draft-duke-quic-version-aliasing-04.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>>.
- [RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <<https://www.rfc-editor.org/info/rfc7301>>.
- [RFC8999] Thomson, M., "Version-Independent Properties of QUIC", RFC 8999, DOI 10.17487/RFC8999, May 2021, <<https://www.rfc-editor.org/info/rfc8999>>.

Appendix A. Changelog

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

A.1. since draft-duke-quic-v2-01

- * Made the final version number TBD.
- * Added ALPN considerations

A.2. since draft-duke-quic-v2-00

- * Added provisional versions for interop
- * Change the v1 Retry Tag secret
- * Change labels to create full key separation

Author's Address

Martin Duke
F5 Networks, Inc.

Email: martin.h.duke@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 is 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.

[illegible]

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

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

Internet Engineering Task Force
Internet-Draft
Intended status: Informational
Expires: 26 April 2022

N. Kuhn
CNES
E. Stephan
Orange
G. Fairhurst
T. Jones
University of Aberdeen
C. Huitema
Private Octopus Inc.
23 October 2021

Transport parameters for 0-RTT connections
draft-kuhn-quic-0rtt-bdp-11

Abstract

QUIC 0-RTT transport features currently focuses on egress traffic optimization. This draft describes a QUIC extension that can be used to improve the performance of ingress traffic.

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 26 April 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
1.1. Notations and terms	4
1.2. Requirements Language	5
2. Safe jump start	5
2.1. Rationale behind the safety guidelines	5
2.2. Rationale #1: Variable network conditions	6
2.3. Rationale #2: Malicious clients	7
2.4. Trade-off between the different solutions	8
2.4.1. Security aspects	8
2.4.2. Interoperability and use-cases	8
2.4.3. Summary	9
3. Safety guidelines	10
4. Implementation considerations	12
4.1. Rationale behind the different implementation options . .	12
4.2. Independent local storage of values	12
4.3. Using NEW_TOKEN frames	13
4.4. BDP Frame	13
4.4.1. BDP Frame Format	13
4.4.2. Extension activation	14
5. Discussion	15
5.1. BDP extension protected as much as initial_max_data . .	15
5.2. Other use-cases	15
5.2.1. Optimizing client's requests	15
5.2.2. Sharing transport information across multiple connections	16
6. Acknowledgments	16
7. IANA Considerations	16
8. Security Considerations	16
9. References	16
9.1. Normative References	16
9.2. Informative References	17
Authors' Addresses	17

1. Introduction

QUIC 0-RTT transport features currently focus on egress traffic optimization. This draft describes a QUIC extension that can be used to improve the performance of ingress traffic.

[RFC9000] mentions that "Generally, implementations are advised to be cautious when using previous values on a new path." This draft proposes a discussion on how using previous values can be achieved in a interoperable manner and how it can be done safely.

When clients resume a session to download a large object, the congestion control algorithms will require time to ramp-up the packet rate as a sequence of Round-Trip Time (RTT)-based increases. This document specifies a method that can improve traffic delivery by allowing a QUIC connection to avoid a the slow process to discover key path parameters including a way to more rapidly grow the congestion window (cwnd):

1. During a previous session, current RTT (`current_rtt`), bottleneck bandwidth (`current_bb`) and current client IP (`current_client_ip`) are stored as `saved_rtt`, `saved_bb` and `saved_client_ip`;
2. When resuming a session to the same IP address, the server can then utilize the `current_rtt` and the `current_bb` to the `saved_rtt` and `saved_bb` of a previous connection.

This method applies to any resumed QUIC session: both `saved_session` and `recon_session` can be a 0-RTT QUIC connection or a 1-RTT QUIC connection.

The current version of this draft considers several possible solutions: (1) the saved parameters are stored at the server; they are not sent to the client; (2) the saved parameters are sent to the client as an encrypted opaque blob; although the client is unable to read the parameters can include this opaque blob in a subsequent request to the server; (3) the saved parameters are sent to the client and the client is notified of their value, but the parameters also include a cryptographic integrity check; the client can include both the parameters and the integrity check in a subsequent request to the server.

None of these possible solutions allow q client to modify the parameters that will be used by the server.

There are several cases where the parameters of a previous session are not appropriate. These include:

- (1) the network conditions have changed and the current capacity is less than the previously estimated bottleneck bandwidth. Using the saved congestion control state would increase congestion;

(2) the network path has changed and the new path is different. Using the saved congestion control state could increase congestion. This case might be accompanied by a change in the RTT or IP address.

(3) a client uses parameters that are no longer appropriate, e.g., to intentionally try to use a CWND larger than appropriate.

This document:

1. proposes guidelines for how to safely apply the previously computed parameters to new sessions;
2. describes different implementation considerations for the proposed method using QUIC;
3. discusses the trade-offs associated with the different implementation solutions.

1.1. Notations and terms

- * IW: Initial Window (e.g., from [RFC6928]);
- * current_iw: Current Initial Window
- * recom_iw: Recommended Initial Window
- * BDP: defined below
- * CWND: the congestion window used by server (maximum number of bytes allowed in flight by the CC)
- * current_bb : Current estimated bottleneck bandwidth
- * saved_bb: Estimated bottleneck bandwidth preserved from a previous connection
- * RTT: Round-Trip Time
- * current_rtt: Current RTT
- * saved_rtt: RTT preserved from a previous connection
- * client_ip : IP address of the client
- * current_client_ip : Current IP address of the client

- * `saved_client_ip` : IP address of the client preserved from a previous connection
- * remembered BDP parameters: a combination of `saved_rtt` and `saved_bb`
- * ITT : Interpacket Transmission Time
- * MSS : Maximum Message Size
- * AEAD : Authenticated Encryption with Associated Data
- * LRU : Least Recently Used

[RFC6349] defines the BDP as follows: "Derived from Round-Trip Time (RTT) and network Bottleneck Bandwidth (BB), the Bandwidth-Delay Product (BDP) determines the Send and Received Socket buffer sizes required to achieve the maximum TCP Throughput." This draft considers the BDP estimated by a server that includes all buffering along the network path. In that sense, the BDP estimated is related to the amount of bytes in flight.

A QUIC connection might not reproduce the procedure detailed in [RFC6349] to measure the BDP. A server might be able to exploit an internal evaluation of the Bottleneck Bandwidth to estimate the BDP.

This document refers to the `saved_bb` and `current_bb` for the previously estimated bottleneck bandwidth. This value can be easily estimated when using a rate-based congestion controller, such as BBR. Other congestion controllers, such as CUBIC or RENO, could estimate the bottleneck bandwidth by utilizing a combination of the `cwnd` and the minimum RTT. This approach could result in over estimating the bottleneck bandwidth and ought to be used with caution.

1.2. Requirements Language

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

2. Safe jump start

2.1. Rationale behind the safety guidelines

The previously measured `saved_rtt` and `saved_bb` SHOULD NOT be used as-is, to avoid potential congestion collapse:

- * Rationale #1: Internet path capacity can change at any time. An Internet method needs to be robust to network conditions that can differ from one session to the next.
- * Rationale #2: Information sent by a malicious client is not relevant. A client could try to convince a server to use a CWND higher than appropriate, to gain an unfair share of capacity for itself or to induce congestion for other flows.

2.2. Rationale #1: Variable network conditions

The server MUST check the validity of the `saved_rtt` and `saved_bb` parameters, whether these are sent by a client or are stored at the server. The following events indicates cases where use of these parameters is inappropriate:

- * IP address changed: If the client changes its IP address (i.e. the `saved_client_ip` is different from the `current_client_ip`), the different address is to be taken as an indication of a different network path. This new path does not necessarily exhibit the same characteristics as the old one. If the server changes its IP address after a migration, it would not be safe to exploit previously estimated parameters.
- * RTT changed: A significant change in RTT might be an indication that the network conditions changed. Since the CC information is directly impacted by the RTT, a significant change in RTT is a strong indication that the previously estimated BDP parameters are likely to not be valid for the current path.
- * Lifetime of the extension: The CC information is temporal. Frequent connections to the same IP address are likely to track changes, but long-term use of previous values are not appropriate.
- * BB over estimation: There are cases where using the `cwnd` would infringe the bottleneck bandwidth. However, at the end of a CC slow start, the value of `cwnd` can be significantly larger than the value, that the CC finally converges to (after a few more rounds). Directly exploiting such value for the bottleneck bandwidth estimation may be inappropriate. One mitigation could be to restrict to only a fraction (e.g., $1/2$) of the previously used `cwnd`; another mitigation might be to calculate the bottleneck bandwidth based on the flight size.

There are different solutions for the variable network conditions:

- * Rationale #1 - Solution #1 : When resuming a session, restore the current_bb and current_rtt from the saved_bb and saved_rtt parameters estimated from a previous connection.
- * Rationale #1 - Solution #2 : When resuming a session, implement a safety check to measure avoid using the saved_bb and saved_rtt parameters to cause congestion over the path. In this case, the current_bb and current_rtt might not be set directly to the saved_bb and saved_rtt: the server might wait for the completion of the safety check before doing so.

Section 3 describes various approaches for Rationale #1 - Solution #2.

2.3. Rationale #2: Malicious clients

The server MUST check the integrity of the saved_rtt and saved_bb parameters received from a client.

There are several solutions to avoid attacks by malicious clients:

- * Rationale #2 - Solution #1 : The server stores a local estimate of the bottleneck bandwidth and RTT parameters as the saved_bb and saved_rtt.
- * Rationale #2 - Solution #2 : The server sends the estimate of the bottleneck bandwidth and RTT parameters to the client as the saved_bb and saved_rtt. This information is encrypted by the server. The client resends the same encrypted information when resuming a connection. The client can neither read nor modify the saved_rtt and saved_bb parameters.
- * Rationale #2 - Solution #3 : The server sends an estimate of the saved_rtt and saved_bb parameters to the client. The information includes an integrity protection check. The client can resend the information when resuming a connection. This allows a client to read, but not modify, the saved_rtt and saved_bb parameters. This might enable a client to decide whether the new parameters are appropriate, based on client-side information about the network conditions or connectivity.

Section 4 describes various implementation approaches for each of these solutions using local storage (Section 4.2 for Rationale #2 - Solution #1), NEW_TOKEN Frame (Section 4.3 for Rationale #2 - Solution #2), BDP extension Frame (Section 4.4 for Rationale #2 - Solution #3).

2.4. Trade-off between the different solutions

This section provides a description of different implementation options and discusses their respective advantages and drawbacks. While there are some discussions for the solutions regarding Rationale #2, the server MUST consider Rationale #1 - Solution #2 and avoid Rationale #1 - Solution #1: the server MUST implement a safety check to measure whether the saved BDP parameters (i.e. saved_rtt and saved_bb) are relevant or check that their usage would not cause excessive congestion over the path.

2.4.1. Security aspects

The client can send information related to the saved_rtt and saved_bb to the server with the BDP Frame extension using either Rationale #2 - Solution #2 or Rationale #2 - Solution #3. However, the server SHOULD NOT trust the client. Indeed, even if 0-RTT packets containing the BDP Frame are encrypted, a client could modify the values within the extension and encrypt the 0-RTT packet. Authentication mechanisms might not guarantee that the values are safe. It is not an easy operation for a client to modify authenticated or encrypted data without this being detected by a server. Modification could be realized by malicious clients. One way to avoid this is for a server to also store the saved_rtt and saved_bb parameters.

A malicious client might modify the saved_bb parameter to convince the server to use a larger CWND than appropriate. Using the algorithms proposed in Section 3, the server may reduce any intended harm and can check that part of the information provided by the client are valid.

Storing the BDP parameters locally at the server reduces the associated risks by allowing the client to transmit information related to the BDP of the path in the case of a malicious client trying to break the encryption mechanism that it had received.

2.4.2. Interoperability and use-cases

If the server stores a resumption ticket for each client to protect against replay on a third party IP, it could also store the IP address (i.e. saved_client_ip) and BDP parameters (i.e. saved_rtt and saved_bb) of the previous session of the client.

In cases where the BDP Frame extension is exploited, the approach of storing the BDP parameters locally at the server can provide a cross-check of the BDP parameters sent by a client. The server can anyway enable a safe jumpstart, but without the BDP Frame extension.

However, the client does not have the choice of accepting to use this or not, and is unable to utilize local knowledge of the network conditions or connectivity.

Storing local values related to the BDP would help in improving the ingress for 0-RTT connections, however, not using a BDP Frame extension could reduce the interest of the approach where (1) the client knows the BDP estimations done at the server, (2) the client decides to accept or reject ingress optimization, (3) the client tunes application level requests.

2.4.3. Summary

As a summary, the approach of local storage of values can be secure and the BDP Frame extension provides more information to the client and more interoperability. The Figure 1 provides a summary of the advantages and drawbacks of each approach.

Rationale	Solution	Advantage	Drawback	Comment
#1 Variable Network	#1 set current_* to saved_*	Ingress optim.	Risks of adding congestion	MUST NOT implement
	#2 Implement safety check	Reduce risks of adding congestion	Negative impact on ingress optim.	MUST implement Section 3
#2 Malicious client	#1 Local storage	Enforced security	Client unable to decide to reject Malicious server could fill client's buffer Limited use-cases	Section 4.2
	#2 NEW_TOKEN	Save resource at server Opaque token protected	Malicious client could change token even if protected	

			Malicious server could fill client's buffer Server may not trust client	Section 4.3
	#3 BDP extension	Extended use-cases Save resource at server Client can read and decide to reject BDP extension protected	Malicious client could change BDP even if protected Server may not trust client	Section 4.4

Figure 1: Comparing solutions

3. Safety guidelines

The safety guidelines are designed to avoid a server adding excessive congestion to an already congested path. The following mechanisms help in fulfilling this objective:

- * The server SHOULD compare the measured transport parameters (in particular `current_rtt`) of the 0-RTT connection with those of the 1-RTT connection (in particular `saved_rtt`);
- * The server SHOULD NOT consider the `saved_bb` parameter when there is any indicated congestion (e.g., loss of packet during the first transmission of data or ECN-CE mark);
- * The server MUST NOT send more than the recommended maximum IW (`recom_iw`) in the first transmission of data. This value could be based on a local understanding of the path characteristics. Knowing the congestion status of the network in closed environments may help in increasing the recommended maximum IW.
- * The server SHOULD NOT store and/or send information related to the previously estimated bottleneck bandwidth (`saved_bb`) (see Section 1.1 for more details on bottleneck bandwidth definition), if this estimation has not been computed after some rounds during the 1-RTT connection. At least, the 1-RTT connection should have reached the congestion avoidance phase.

The proposed mechanisms SHOULD be limited by any rate-limitation mechanisms of QUIC, such as flow control mechanisms or amplification attack prevention. In particular, it may be necessary to issue proactive MAX_DATA frames to increase the flow control limits of a connection. In particular, the maximum number of packets that can be sent without acknowledgment needs to be chosen to avoid the creation and the increase of congestion for the path.

This extension should not provide an opportunity for the current connection to be a vector of an amplification attack. The address validation process, used to prevent amplification attacks, SHOULD be performed [RFC9000].

The following mechanisms could be implemented:

* Exploit a standard IW:

1. The server sends the first data packet using the IW - this is a safe starting point for any path where there is no path information or where there is no congestion state. This avoids adding excessive congestion to the path;
2. If the reception of IW exhibits characteristics that resemble those of a recent previous session from the client (i.e. $\text{current_rtt} < 1.2 * \text{saved_rtt}$ and all data was acknowledged without reported congestion), the method permits the sender to consider the saved_bb as an input to adapt current_bb to rapidly determine a new safe rate;
3. The sender needs to avoid a burst of packets resulting from a step-increase in the congestion window [RFC9000]. Pacing the packets as a function of the current_rtt can provide this additional safety during the period in which the CWND is increased by the method.

* Identify a relevant pacing rhythm:

- The server estimates the pacing rhythm using saved_rtt and saved_bb. The Interpacket Transmission Time (ITT) is determined by the ratio between the current Maximum Message Size (MSS) for packets and the ratio between the saved_bb and saved_rtt. A tunable safety margin might be introduced to avoid sending more than a recommended maximum IW (recom_iw):
 - o $\text{current_iw} = \min(\text{recom_iw}, \text{saved_bb})$
 - o $\text{ITT} = \text{MSS} / (\text{current_iw} / \text{saved_rtt})$

- When the IW is acknowledged, the server falls back to a standard slow-start mechanism.
- * Tune slow-start mechanisms: After transport parameters are set to a previously estimated bottleneck bandwidth, if slow-start mechanisms continue, the sender can overshoot the bottleneck capacity. This can occur even if the safety check described in this section is implemented.
- For NewReno and CUBIC, it is recommended to exit slow-start and enter in congestion avoidance phase.
- For BBR, it is recommended to move to the "probe bandwidth" state.

This follows the idea of [RFC4782],
[I-D.irtf-iccr-g-sallantin-initial-spreading] and [CONEXT15].

4. Implementation considerations

4.1. Rationale behind the different implementation options

The NewSessionTickets messages of TLS offer a solution. The idea would have been to add a 'bdp_metada' field in the NewSessionTickets that the client could read. The sole extension currently defined in TLS1.3 that can be seen by the client is max_early_data_size (see section 4.6.1 of [RFC8446]). However, in the general design of QUIC, TLS sessions are managed by the TLS stacks.

Three distinct approaches are presented: sending an opaque blob to the client that it may return to the server for a future connection (see Section 4.3), enable a local storage of BDP related values (see Section 4.2) and a BDP Frame extension (see Section 4.4).

4.2. Independent local storage of values

This approach independently lets both a client and a server remember their BDP parameters:

- * During a 1-RTT session, the endpoint stores the RTT (as the saved_rtt) and bottleneck bandwidth (as the saved_bb) together with the session resume ticket. The client can also store the IP address of the server.
- * The server maintains a table of previously issued tickets, indexed by the random ticket identifier that is used to guarantee uniqueness of the Authenticated Encryption with Associated Data (AEAD) encryption. Old tokens are removed from the table using

the Least Recently Used (LRU) logic. For each ticket identifier, the table holds the RTT and bottleneck bandwidth (i.e. `saved_rtt` and `saved_bb`), and also the IP address of the client (i.e. `saved_client_ip`).

During the 0-RTT session, the endpoint waits for the first RTT measurement from the peer's IP address. This is used to verify that the `current_rtt` has not significantly changed from the `saved_rtt`, and hence is an indication that the BDP information is appropriate to the path that is currently being used.

If this RTT is confirmed (e.g. `current_rtt < 1.2*saved_rtt`, the endpoint also verifies that an initial window of data has been acknowledged without requiring retransmission. This second check detects a path with significant incipient congestion (i.e. where it would not be safe to update the CWND based on the `saved_bb`). In practice, this could be realized by a proportional increase in the CWND, where the increase is $(\text{saved_bb}/\text{IW}) * \text{proportion_of_IW_currently_ACKed}$.

This solution does not allow the client to refuse the exploitation of the BDP parameters. If the server does not want to store the metrics from previous connections, an equivalent of the `tcp_no_metrics_save` for QUIC may be necessary. This option could be negotiated that allows a client to choose whether to use the saved information.

4.3. Using NEW_TOKEN frames

Using NEW_TOKEN Frames, the server could send a token to the client through a NEW_TOKEN Frame. The token is an opaque blob and the client can not read its content (see section 19.7 of [RFC9000]). The client sends the received token in the header of an Initial packet for a later connection.

4.4. BDP Frame

This section describes the use of a new Frame, the BDP Frame. The BDP Frame MUST be contained in 0-RTT packets, if sent by the client. The BDP Frame MUST be contained in 1-RTT packets, if sent by the server. The BDP Frame MUST be considered by congestion control and its data is not be limited by flow control limits. The server MAY send multiple BDP Frames in both 1-RTT and 0-RTT connections. The client can send BDP Frames during 1-RTT and 0-RTT connections.

4.4.1. BDP Frame Format

A BDP Frame is formatted as shown in Figure 2.

```
BDP Frame {  
    Type (i) = 0xXXX,  
    Lifetime (i),  
    Saved BB (i),  
    Saved RTT (i),  
    Saved IP length (i),  
    Saved IP (...)  
}
```

Figure 2: BDP Frame Format

A BDP Frame contains the following fields:

- * Lifetime (extension_lifetime): The extension_lifetime is a value in milliseconds, encoded as a variable length integer. This follows the idea of NewSessionTicket of TLS [RFC8446]. This represents the validity in time of this extension.
- * Saved BB (saved_bb): The saved_bb is a value in bytes, encoded as a variable length integer. The bottleneck bandwidth estimated for the previous connection by the server. Using the previous values of bytes_in_flight defined in [RFC9002] can result in overshoot of the bottleneck capacity and is not advised.
- * Saved RTT (saved_rtt): The saved_rtt is a value in milliseconds, encoded as a variable length integer. This could be set to the minimum RTT (min_rtt). The saved_rtt can be set to min_rtt. NOTE: The min_rtt defined in [RFC9002], does not track a decreasing RTT: therefore min_rtt reported might be larger than the actual minimum RTT measured during the 1-RTT connection.
- * Saved IP length (saved_ip_length) : The length of the IP address set to either 4 (IPv4) or 16 (IPv6).
- * Saved IP (saved_client_ip) : The saved_client_ip could be set to the IP address of the client.

4.4.2. Extension activation

The client can accept the transmission of BDP Frames from the server by using the enable_bdp transport extension.

enable_bdp (0xTBD): in the 1-RTT connection, the client indicates to the server that it wishes to receive BDP extension Frames for improving ingress of 0-RTT connection. The default value is 0. Values strictly above 3 are invalid, and receipt of these values MUST be treated as a connection error of type TRANSPORT_PARAMETER_ERROR.

- * 0: Default value. If the client does not send this parameter, the server considers that the client does not support or does not wish to activate the BDP extension.
- * 1: The client indicates to the server that it wishes to receive BDP Frame and activates the ingress optimization for the 0-RTT connection.
- * 2: The client indicates that it does not wish to receive BDP Frames but activates ingress optimization.
- * 3: The client indicates that it wishes to receive BDP Frames but does not activate ingress optimization.

This Transport Parameter is encoded as per Section 18 of [RFC9000].

5. Discussion

5.1. BDP extension protected as much as initial_max_data

The BDP metadata parameters are measured by the server during a previous connection. The BDP extension is protected by the mechanism that protects the exchange of the 0-RTT transport parameters. For version 1 of QUIC, the BDP extension is protected using the mechanism that already protects the "initial_max_data" parameter. This is defined in sections 4.5 to 4.7 of [RFC9001]. This provides a way for the server to verify that the parameters proposed by the client are the same as those that the server sent to the client during the previous connection.

5.2. Other use-cases

5.2.1. Optimizing client's requests

When using Dynamic Adaptive Streaming over HTTPS (DASH), clients might encounter issues in knowing the available path capacity or DASH can encounter issues in reaching the best available video playback quality. The client requests could then be adapted and specific traffic could utilize information from the path characteristics (such as encouraging the client to increase the quality of video chunks, to fill the buffers and avoid video blocking or to send high quality adds).

In other cases, applications could provide additional services if clients can know the server estimation of the path characteristics.

5.2.2. Sharing transport information across multiple connections

There can be benefit in sharing transport information across multiple connections. [I-D.ietf-tcpm-2140bis] considers the sharing of transport parameters between TCP connections originating from the same host. The proposal in this document has the advantage of storing server-generated information at the client and not requiring the server to retain additional state for each client.

6. Acknowledgments

The authors would like to thank Gabriel Montenegro, Patrick McManus, Ian Swett, Igor Lubashev, Robin Marx, Roland Bless and Franklin Simo for their fruitful comments on earlier versions of this document.

7. IANA Considerations

TBD: Text is required to register the BDP Frame and the enable_bdp transport parameter. Parameters are registered using the procedure defined in [RFC9000].

8. Security Considerations

Security considerations are discussed in Section 5 and in Section 3.

9. References

9.1. Normative References

- [I-D.ietf-tcpm-2140bis]
Touch, J., Welzl, M., and S. Islam, "TCP Control Block Interdependence", Work in Progress, Internet-Draft, draft-ietf-tcpm-2140bis-11, 12 April 2021, <<https://www.ietf.org/archive/id/draft-ietf-tcpm-2140bis-11.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>>.
- [RFC4782] Floyd, S., Allman, M., Jain, A., and P. Sarolahti, "Quick-Start for TCP and IP", RFC 4782, DOI 10.17487/RFC4782, January 2007, <<https://www.rfc-editor.org/info/rfc4782>>.

- [RFC6349] Constantine, B., Forget, G., Geib, R., and R. Schrage, "Framework for TCP Throughput Testing", RFC 6349, DOI 10.17487/RFC6349, August 2011, <<https://www.rfc-editor.org/info/rfc6349>>.
- [RFC6928] Chu, J., Dukkupati, N., Cheng, Y., and M. Mathis, "Increasing TCP's Initial Window", RFC 6928, DOI 10.17487/RFC6928, April 2013, <<https://www.rfc-editor.org/info/rfc6928>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/info/rfc8446>>.
- [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>>.
- [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/info/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/info/rfc9002>>.

9.2. Informative References

- [CONEXT15] Li, Q., Dong, M., and P B. Godfrey, "Halfback: Running Short Flows Quickly and Safely", ACM CoNEXT , 2015.
- [I-D.irtf-iccr-g-sallantin-initial-spreading]
Sallantin, R., Baudoin, C., Arnal, F., Dubois, E., Chaput, E., and A. Beylot, "Safe increase of the TCP's Initial Window Using Initial Spreading", Work in Progress, Internet-Draft, draft-irtf-iccr-g-sallantin-initial-spreading-00, 15 January 2014, <<https://www.ietf.org/archive/id/draft-irtf-iccr-g-sallantin-initial-spreading-00.txt>>.

Authors' Addresses

Nicolas Kuhn
CNES

Email: nicolas.kuhn.ietf@gmail.com

Emile Stephan
Orange

Email: emile.stephan@orange.com

Godred Fairhurst
University of Aberdeen
Department of Engineering
Fraser Noble Building
Aberdeen

Email: gorry@erg.abdn.ac.uk

Tom Jones
University of Aberdeen
Department of Engineering
Fraser Noble Building
Aberdeen

Email: tom@erg.abdn.ac.uk

Christian Huitema
Private Octopus Inc.

Email: huitema@huitema.net

QUIC Working Group
Internet-Draft
Intended status: Standards Track
Expires: 28 April 2022

Y. Liu
Y. Ma
Alibaba Inc.
Q. De Coninck
O. Bonaventure
UCLouvain
C. Huitema
Private Octopus Inc.
M. Kuehlewind, Ed.
Ericsson
25 October 2021

Multipath Extension for QUIC
draft-lmbdhk-quic-multipath-00

Abstract

This document specifies a multipath extension for the QUIC protocol to enable the simultaneous usage of multiple paths for a single 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 (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/mirjak/draft-lmbdhk-quic-multipath>.

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 28 April 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	3
1.1. Conventions and Definitions	4
2. Handshake Negotiation and Transport Parameter	5
3. Path Setup and Removal	6
3.1. Path Initiation	7
3.2. Path Close	7
3.2.1. Use PATH_ABANDON Frame to Close a Path	7
3.2.2. Effect of RETIRE_CONNECTION_ID Frame	8
3.2.3. Idle Timeout	9
3.3. Path States	9
4. Congestion Control	11
5. Computing Path RTT	11
6. Packet Scheduling	12
7. Packet Number Space and Use of Connection ID	13
7.1. Using One Packet Number Space	13
7.1.1. Sending Acknowledgements and Handling Ranges	14
7.2. Using Multiple Packet Number Spaces	15
7.2.1. Packet Protection for QUIC Multipath	15
7.2.2. Key Update for QUIC Multipath	16
8. Examples	17
8.1. Path Establishment	17
8.2. Path Closure	18
9. Implementation Considerations	19
10. New Frames	19
10.1. PATH_ABANDON Frame	19
10.2. ACK_MP Frame	21
11. Error Codes	22
12. IANA Considerations	22
13. Security Considerations	23
14. Contributors	23

15. Acknowledgments	23
16. References	23
16.1. Normative References	23
16.2. Informative References	24
Authors' Addresses	25

1. Introduction

This document specifies an extension to QUIC v1 [QUIC-TRANSPORT] to enable the simultaneous usage of multiple paths for a single connection.

This proposal is based on several basic design points:

- * Re-use as much as possible mechanisms of QUIC-v1. In particular this proposal uses path validation as specified for QUIC v1 and aims to re-use as much as possible of QUIC's connection migration.
- * Use the same packet header formats as QUIC v1 to avoid the risk of packets being dropped by middleboxes (which may only support QUIC v1)
- * Congestion Control, RTT measurements and PMTU discovery should be per-path (following [QUIC-TRANSPORT])
- * A path is determined by the 4-tuple of source and destination IP address as well as source and destination port. Therefore there can be at most one active paths/connection ID per 4-tuple.

The path management specified in section 9 of [QUIC-TRANSPORT] fulfills multiple goals: it directs a peer to switch sending through a new preferred path, and it allows the peer to release resources associated with the old path. Multipath requires several changes to that mechanism:

- * Allow simultaneous transmission of non probing frames on multiple paths.
- * Continue using an existing path even if non-probing frames have been received on another path.
- * Manage the removal of paths that have been abandoned.

As such this extension specifies a departure from the specification of path management in section 9 of [QUIC-TRANSPORT] and therefore requires negotiation between the two endpoints using a new transport parameter, as specified in Section 2.

This proposal supports the negotiation of either the use of one packet number space for all paths or the use of separate packet number spaces per path. While separate packet number spaces allow for more efficient ACK encoding, especially when paths have highly different latencies, this approach requires the use of a connection ID. Therefore use of a single number space can be beneficial in highly constrained networks that do not benefit from exposing the connection ID in the header. While both approaches are supported by the specification in this version of the document, the intention for the final publication of a multipath extension for QUIC is to choose one option in order to avoid incompatibility. More evaluation and implementation experience is needed to select one approach before final publication. Some discussion about pros and cons can be found here: https://github.com/mirjak/draft-lmbdtk-quic-multipath/blob/master/presentations/PacketNumberSpace_s.pdf

As currently defined in this version of the draft the use of multiple packet number spaces requires the use of connection IDs in both directions. Today's deployments often only use destination connection ID when sending packets from the client to the server as this addresses the most important use cases for migration, like NAT rebinding or mobility events. Further discussion and work is required to evaluate if the use of multiple packet number spaces could be supported as well when the connection ID is only present in one direction.

This proposal does not cover address discovery and management. Addresses and the actual decision process to setup or tear down paths are assumed to be handled by the application that is using the QUIC multipath extension. Further, this proposal only specifies a simple basic packet scheduling algorithm in order to provide some basic implementation guidance. However, more advanced algorithms as well as potential extensions to enhance signaling of the current path state are expected as future work.

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.

We assume that the reader is familiar with the terminology used in [QUIC-TRANSPORT]. In addition, we define the following terms:

- * Path Identifier (Path ID): An identifier that is used to identify a path in a QUIC connection at an endpoint. Path Identifier is used in multi-path control frames (etc. PATH_ABANDON frame) to identify a path. By default, it is defined as the sequence number of the destination Connection ID used for sending packets on that particular path, but alternative definitions can be used if the length of that connection ID is zero.
- * Packet Number Space Identifier (PN Space ID): An identifier that is used to distinguish packet number spaces for different paths. It is used in 1-RTT packets and ACK_MP frames. Each node maintains a list of "Received Packets" for each of the CID that it provided to the peer, which is used for acknowledging packets received with that CID.

The difference between Path Identifier and Packet Number Space Identifier, is that the Path Identifier is used in multipath control frames to identify a path, and the Packet Number Space Identifier is used in 1-RTT packets and ACK_MP frames to distinguish packet number spaces for different paths. Both identifiers have the same value, which is the sequence number of the connection ID, if a non-zero connection ID is used. If the connection ID is zero length, the Packet Number Space Identifier is 0, while the Path Identifier is selected on path establishment.

2. Handshake Negotiation and Transport Parameter

This extension defines a new transport parameter, used to negotiate the use of the multipath extension during the connection handshake, as specified in [QUIC-TRANSPORT]. The new transport parameter is defined as follow:

- * name: enable_multipath (TBD - experiments use 0xbabf)
- * value: 0 (default) for disabled. Endpoints use 2-bits in the value field for negotiating one or more PN spaces, available option value for client and server are listed in Table 1 :

Client Option	Definition	Allowed server responses
0x0	don't support multi-path	0x0
0x1	only support one PN space for multi-path	0x0 or 0x1
0x2	only support multiple PN spaces for multi-path	0x0 or 0x2
0x3	support both one PN space and multiple PN space	0x0, 0x1 or 0x2

Table 1: Available value for enable_multipath

If the peer does not carry the enable_multipath transport parameter, which means the peer does not support multipath, endpoint MUST fallback to [QUIC-TRANSPORT] with single path and MUST NOT use any frame or mechanism defined in this document. If endpoint receives unexpected value for the transport parameter "enable_multipath", it MUST treat this as a connection error of type MP_CONNECTION_ERROR and close the connection.

Note that the transport parameter "active_connection_id_limit" [QUIC-TRANSPORT] limits the number of usable Connection IDs, and also limits the number of concurrent paths. For the QUIC multipath extension this limit even applies when no connection ID is exposed in the QUIC header.

3. Path Setup and Removal

After completing the handshake, endpoints have agreed to enable multipath feature and can start using multiple paths. This document does not discuss when a client decides to initiate a new path. We delegate such discussion in separate documents.

This proposal adds one multi-path control frame for path management:

- * PATH_ABANDON frame for the receiver side to abandon the path
Section 10.1

All the new frames are sent in 1-RTT packets [QUIC-TRANSPORT].

3.1. Path Initiation

When the multipath option is negotiated, clients that want to use an additional path MUST first initiate the Address Validation procedure with PATH_CHALLENGE and PATH_RESPONSE frames described in Section 8 of [QUIC-TRANSPORT]. After receiving packets from the client on the new paths, the servers MAY in turn attempt to validate these paths using the same mechanisms.

If validation succeed, the client can send non-probing, 1-RTT packets on the new paths. In contrast with the specification in section 9 of [QUIC-TRANSPORT], the server MUST NOT assume that receiving non-probing packets on a new path indicates an attempt to migrate to that path. Instead, servers SHOULD consider new paths over which non-probing packets have been received as available for transmission.

3.2. Path Close

Each endpoint manages the set of paths that are available for transmission. At any time in the connection, each endpoint can decide to abandon one of these paths, following for example changes in local connectivity or changes in local preferences. After an endpoint abandons a path, the peer will not receive any more non-probing packets on that path.

An endpoint that wants to close a path SHOULD NOT rely on implicit signals like idle time or packet losses, but instead SHOULD use explicit request to terminate path by sending the PATH_ABANDON frame (see Section 10.1).

3.2.1. Use PATH_ABANDON Frame to Close a Path

Both endpoints, namely the client and the server, can close a path, by sending PATH_ABANDON frame (see Section 10.1) which abandons the path with a corresponding Path Identifier. Once a path is marked as "abandoned", it means that the resources related to the path, such as the used connection IDs, can be released. However, information related to data delivered over that path SHOULD not be released immediately as acknowledgments can still be received or other frames that also may trigger retransmission of data on another path.

The endpoint sending the PATH_ABANDON frame SHOULD consider a path as abandoned when the packet that contained the PATH_ABANDON frame is acknowledged. When releasing resources of a path, the endpoint SHOULD send a RETIRE_CONNECTION_ID frame for the connection IDs used on the path, if any.

The receiver of a PATH_ABANDON frame SHOULD NOT release its resources immediately but SHOULD wait for the receive of the RETIRE_CONNECTION_ID frame for the used connection IDs or 3 RTOs.

Usually it is expected that the PATH_ABANDON frame is used by the client to indicate to the server that path conditions have changed such that the path is or will be not usable anymore, e.g. in case of an mobility event. The PATH_ABANDON frame therefore indicates to the receiving peer that the sender does not intend to send any packets on that path anymore but also recommends to the receiver that no packets should be sent in either direction. The receiver of an PATH_ABANDON frame MAY also send an PATH_ABANDON frame to signal its own willingness to not send any packet on this path anymore.

If connection IDs are used, PATH_ABANDON frames can be sent on any path, not only the path that is intended to be closed. Thus a connection can be abandoned even if connectivity on that path is already broken. If no connection IDs are used and the PATH_ABANDON frame has to sent on the path that is intended to be closed, it is possible that the packet containing the PATH_ABANDON frame or the packet containing the ACK for the PATH_ABANDON frame cannot be received anymore and the endpoint might need to rely on an idle time out to close the path, as described in Section Section 3.2.3.

Retransmittable frames, that have previously been send on the abandoned path and are considered lost, SHOULD be retransmitted on a different path.

If a PATH_ABANDON frame is received for the only active path of a QUIC connection, the receiving peer SHOULD send a CONNECTION_CLOSE frame and enters the closing state. If the client received a PATH_ABANDON frame for the last open path, it MAY instead try to open a new path, if available, and only initiate connection closure if path validation fails or a CONNECTION_CLOSE frame is received from the server. Similarly the server MAY wait for a short, limited time such as one RTO if a path probing packet is received on a new path before sending the CONNECTION_CLOSE frame.

3.2.2. Effect of RETIRE_CONNECTION_ID Frame

Receiving a RETIRE_CONNECTION_ID frame causes the endpoint to discard the resources associated with that connection ID. If the connection ID was used by the peer to identify a path from the peer to this endpoint, the resources include the list of received packets used to send acknowledgements. The peer MAY decide to keep sending data using the same IP addresses and UDP ports previously associated with the connection ID, but MUST use a different connection ID when doing so.

3.2.3. Idle Timeout

[QUIC-TRANSPORT] allows for closing of connections if they stay idle for too long. The connection idle timeout in multipath QUIC is defined as "no packet received on any path for the duration of the idle timeout". When only one path is available, servers MUST follow the specifications in [QUIC-TRANSPORT].

When more than one path is available, servers shall monitor the arrival of non-probing packets on the available paths. Servers SHOULD stop sending traffic on paths through where no non-probing packet was received in the last 3 path RTTs, but MAY ignore that rule if it would disqualify all available paths. Server MAY release the resource associated with paths for which no non-probing packet was received for a sufficiently long path-idle delay, but SHOULD only release resource for the last available path if no traffic is received for the duration of the idle timeout, as specified in section 10.1 of [QUIC-TRANSPORT]. This means if all paths remain idle for the idle timeout, the connection is implicitly closed.

Server implementations need to select the sub-path idle timeout as a trade-off between keeping resources, such as connection IDs, in use for an excessive time or having to promptly reestablish a path after a spurious estimate of path abandonment by the client.

3.3. Path States

Figure 1 shows the states that an endpoint's path can have.

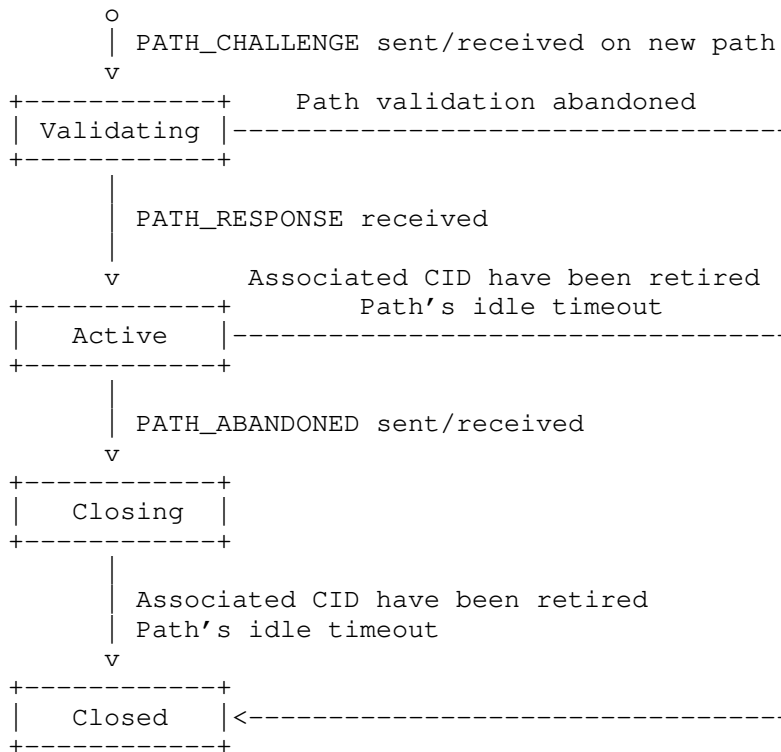


Figure 1: States of a path

In non-final states, hosts have to track the following information.

- * **Associated 4-tuple:** The tuple (source IP, source port, destination IP, destination port) used by the endhost to send packets over the path.
- * **Associated Destination Connection ID:** The Connection ID used to send packets over the path.

If multiple packet number spaces are used over the connection, hosts MUST also track the following information.

- * **Path Packet Number Space:** The endpoint maintains a separate packet number for sending and receiving packets over this path. Packet number considerations described in [QUIC-TRANSPORT] apply within the given path.

In the "Active" state, hosts MUST also track the following information.

- * Associated Source Connection ID: The Connection ID used to receive packets over the path.

A path in the "Validating" state performs path validation as described in Section 8.2 of [QUIC-TRANSPORT]. An endhost should not send non-probing frames on a path in "Validating" state, as it has no guarantee that packets will actually reach the peer.

The endhost can use all the paths in the "Active" state, provided that the congestion control and flow control currently allow sending of new data on a path.

In the "Closing" state, the endhost SHOULD NOT send packets on this path anymore, as there is no guarantee that the peer can still map the packets to the connection. The endhost SHOULD wait for the acknowledgment of the PATH_ABANDONED frame before moving the path to the "Closed" state to ensure a graceful termination of the path.

When a path reaches the "Closed" state, the endhost releases all the path's associated resources. Consequently, the endhost is not able to send nor receive packets on this path anymore.

4. Congestion Control

Senders MUST manage per-path congestion status, and MUST NOT send more data on a given path than congestion control on that path allows. This is already a requirement of [QUIC-TRANSPORT].

When a Multipath QUIC connection uses two or more paths, there is no guarantee that these paths are fully disjoint. When two (or more paths) share the same bottleneck, using a standard congestion control scheme could result in an unfair distribution of the bandwidth with the multipath connection getting more bandwidth than competing single paths connections. Multipath TCP uses the LIA congestion control scheme specified in [RFC6356] to solve this problem. This scheme can immediately be adapted to Multipath QUIC. Other coupled congestion control schemes have been proposed for Multipath TCP such as [OLIA].

5. Computing Path RTT

Acknowledgement delays are the sum of two one-way delays, the delay on the packet sending path and the delay on the return path chosen for the acknowledgements. When different paths have different characteristics, this can cause acknowledgement delays to vary widely. Consider for example multipath transmission using both a terrestrial path, with a latency of 50ms in each direction, and a geostationary satellite path, with a latency of 300ms in both directions. The acknowledgement delay will depend on the combination

of paths used for the packet transmission and the ACK transmission, as shown in Table 2.

ACK Path \ Data path	Terrestrial	Satellite
Terrestrial	100ms	350ms
Satellite	350ms	600ms

Table 2: Example of ACK delays using multiple paths

Using the default algorithm specified in [QUIC-RECOVERY] would result in suboptimal performance, computing average RTT and standard deviation from series of different delay measurements of different combined paths. At the same time, early tests showed that it is desirable to send ACKs through the shortest path, because a shorter ACK delay results in a tighter control loop and better performances. The tests also showed that it is desirable to send copies of the ACKs on multiple paths, for robustness if a path experiences sudden losses.

An early implementation mitigated the delay variation issue by using time stamps, as specified in [QUIC-Timestamp]. When the timestamps are present, the implementation can estimate the transmission delay on each one-way path, and can then use these one way delays for more efficient implementations of recovery and congestion control algorithms.

If timestamps are not available, implementations could estimate one way delays using statistical techniques. For example, in the example shown in Table 1, implementations can use "same path" measurements to estimate the one way delay of the terrestrial path to about 50ms in each direction, and that of the satellite path to about 300ms. Further measurements can then be used to maintain estimates of one way delay variations, using logical similar to Kalman filters. But statistical processing is error-prone, and using time stamps provides more robust measurements.

6. Packet Scheduling

The transmission of QUIC packets on a regular QUIC connection is regulated by the arrival of data from the application and the congestion control scheme. QUIC packets can only be sent when the congestion window of at least one path is open.

Multipath QUIC implementations also need to include a packet scheduler that decides, among the paths whose congestion window is open, the path over which the next QUIC packet will be sent. Many factors can influence the definition of these algorithms and their precise definition is outside the scope of this document. Various packet schedulers have been proposed and implemented, notably for Multipath TCP. A companion draft [I-D.bonaventure-iccr-g-schedulers] provides several general-purpose packet schedulers depending on the application goals.

7. Packet Number Space and Use of Connection ID

If the connection ID is present (non-zero length) in the packet header, the connection ID is used to identify the path. If no connection ID is present, the 4 tuple identifies the path. The initial path that is used during the handshake (and multipath negotiation) has the path ID 0 and therefore all 0-RTT packets are also tracked and processed with the path ID 0. For 1-RTT packets the path ID is the sequence number of the Destination Connection ID present in the packet header, as defined in Section 5.1.1 of [QUIC-TRANSPORT], or also 0 if the Connection ID is zero-length.

If non-zero-length Connection IDs are used, an endpoint MUST use different Connection IDs on different paths. Still, the receiver may observe the same Connection ID used on different 4-tuples due to, e.g., NAT rebinding. In such case, the receiver reacts as specified in Section 9.3 of [QUIC-TRANSPORT].

Acknowledgements of Initial and Handshake packets MUST be carried using ACK frames, as specified in [QUIC-TRANSPORT]. The ACK frames, as defined in [QUIC-TRANSPORT], do not carry path identifiers. If for any reason ACK frames are received in 1-RTT packets while the state of multipath negotiation is ambiguous, they MUST be interpreted as acknowledging packets sent on path 0.

7.1. Using One Packet Number Space

If the multipath option is negotiated to use one packet number space for all paths, the packet sequence numbers are allocated from the common number space, so that, for example, packet number N could be sent on one path and packet number N+1 on another.

ACK frames report the numbers of packets that have been received so far, regardless of the path on which they have been received. That means the senders needs to maintain an association between sent packet numbers and the path over which these packets were sent. This is necessary to implement per path congestion control.

When a packet is acknowledged, the state of the congestion control MUST be updated for the path where the acknowledged packet was originally sent. The RTT is calculated based on the delay between the transmission of that packet and its first acknowledgement (see Section 5) and is used to update the RTT statistics for the sending path.

Also loss detection MUST be adapted to allow for different RTTs on different paths. For example, timer computations should take into account the RTT of the path on which a packet was sent. Detections based on packet numbers shall compare a given packet number to the highest packet number received for that path.

7.1.1. Sending Acknowledgements and Handling Ranges

If senders decide to send packets on paths with different transmission delays, some packets will very likely be received out of order. This will cause the ACK frames to carry multiple ranges of received packets. The large number of range increases the size of ACK frames, causing transmission and processing overhead.

The size and overhead of the ACK frames can be controlled by the combination of one or several of the following:

- * Not transmitting again ACK ranges that were present in an ACK frame acknowledged by the peer.
- * Delay acknowledgements to allow for arrival of "hole filling" packets.
- * Limit the total number of ranges sent in an ACK frame.
- * Limiting the number of transmissions of a specific ACK range, on the assumption that a sufficient number of transmissions almost certainly ensures reception by the peer.
- * Send multiple messages for a given path in a single socket operation, so that a series of packets sent from a single path uses a series of consecutive sequence numbers without creating holes.

7.2. Using Multiple Packet Number Spaces

If the multipath option is enabled with a value of 2, each path has its own packet number space for transmitting 1-RTT packets and a new ACK frame format is used as specified in Section 10.2. Compared to the QUIC v1 ACK frame, the MP_ACK frames additionally contains a Packet Number Space Identifier (PN Space ID). The PN Space ID used to distinguish packet number spaces for different paths and is simply derived from the sequence number of Destination Connection ID. Therefore, the packet number space for 1-RTT packets can be identified based on the Destination Connection ID in each packets.

As soon as the negotiation of multipath support with value 2 is completed, endpoints SHOULD use ACK_MP frames instead of ACK frames for acknowledgements of 1-RTT packets on path 0, as well as for 0-RTT packets that are acknowledged after the handshake concluded.

Following [QUIC-TRANSPORT], each endpoint uses NEW_CONNECTION_ID frames to issue usable connections IDs to reach it. Before an endpoint adds a new path by initiating path validation, it MUST check whether at least one unused Connection ID is available for each side.

If the transport parameter "active_connection_id_limit" is negotiated as N, the server provided N Connection IDs, and the client is already actively using N paths, the limit is reached. If the client wants to start a new path, it has to retire one of the established paths.

ACK_MP frame Section 10.2 can be returned via either a different path, or the same path identified by the Path Identifier, based on different strategies of sending ACK_MP frames.

Using multiple packet number spaces requires changes in the way AEAD is applied for packet protection, as explained in Section 7.2.1, and tighter constraints for key updates, as explained in Section 7.2.2.

7.2.1. Packet Protection for QUIC Multipath

Packet protection for QUIC v1 is specified in Section 5 of [QUIC-TLS]. The general principles of packet protection are not changed for QUIC Multipath. No changes are needed for setting packet protection keys, initial secrets, header protection, use of 0-RTT keys, receiving out-of-order protected packets, receiving protected packets, or retry packet integrity. However, the use of multiple number spaces for 1-RTT packets requires changes in AEAD usage.

Section 5.3 of [QUIC-TLS] specifies AEAD usage, and in particular the use of a nonce, N , formed by combining the packet protection IV with the packet number. If multiple packet number spaces are used, the packet number alone would not guarantee the uniqueness of the nonce.

In order to guarantee the uniqueness of the None, the nonce N is calculated by combining the packet protection IV with the packet number and with the path identifier.

The path ID for 1-RTT packets is the sequence number of [QUIC-TRANSPORT], or zero if the Connection ID is zero-length. Section 19 of [QUIC-TRANSPORT] encodes the Connection ID Sequence Number as a variable-length integer, allowing values up to $2^{62}-1$; in this specification a range of less than $2^{32}-1$ values MUST be used before updating the packet protection key.

To calculate the nonce, a 96 bit path-and-packet-number is composed of the 32 bit Connection ID Sequence Number in byte order, two zero bits, and the 62 bits of the reconstructed QUIC packet number in network byte order. If the IV is larger than 96 bits, the path-and-packet-number is left-padded with zeros to the size of the IV. The exclusive OR of the padded packet number and the IV forms the AEAD nonce.

For example, assuming the IV value is 6b26114b9cba2b63a9e8dd4f, the connection ID sequence number is 3, and the packet number is aead, the nonce will be set to 6b2611489cba2b63a9e873e2.

7.2.2. Key Update for QUIC Multipath

The Key Phase bit update process for QUIC v1 is specified in Section 6 of [QUIC-TLS]. The general principles of key update are not changed in this specification. Following QUIC v1, the Key Phase bit is used to indicate which packet protection keys are used to protect the packet. The Key Phase bit is toggled to signal each subsequent key update. Because of network delays, packets protected with the older key might arrive later than the packets protected with the new key. Therefore, the endpoint needs to retain old packet keys to allow these delayed packets to be processed and it must distinguish between the new key and the old key. In QUIC V1, this is done using packet numbers so that the rule is made simple: Use the older key if packet number is lower than any packet number frame the current key phase.

When using multiple packet number spaces on different paths, some care is needed when initiating the Key Update process, as different paths use different packet number spaces but share a single key. When a key update is initiated on one path, packets sent to another

path needs to know when the transition is complete. Otherwise, it is possible that the other paths send packets with the old keys, but skip sending any packets in the current key phase and directly jump to sending packet in the next key phase. When that happens, as the endpoint can only retain two sets of packet protection keys with the 1-bit Key Phase bit, the other paths cannot distinguish which key should be used to decode received packets, which results in a key rotation synchronization problem.

To address such a synchronization issue, if key update is initialized on one path, the sender SHOULD send at least one packet with the new key on all active paths. Further, an endpoint MUST NOT initiate a subsequent key update until a packet with the current key has been acknowledged on each path.

Following Section 5.4 of [QUIC-TLS], the Key Phase bit is protected, so sending multiple packets with Key Phase bit flipping at the same time should not cause linkability issue.

8. Examples

8.1. Path Establishment

Figure 2 illustrates an example of new path establishment using multiple packet number spaces.

Client	Server
(Exchanges start on default path)	
1-RTT[]: NEW_CONNECTION_ID[C1, Seq=1] -->	
	<-- 1-RTT[]: NEW_CONNECTION_ID[S1, Seq=1]
	<-- 1-RTT[]: NEW_CONNECTION_ID[S2, Seq=2]
...	
(starts new path)	
1-RTT[0]: DCID=S2, PATH_CHALLENGE[X] -->	
	Checks AEAD using nonce(CID sequence 2, PN 0)
<-- 1-RTT[0]: DCID=C1, PATH_RESPONSE[X], PATH_CHALLENGE[Y],	
	ACK_MP[Seq=2,PN=0]
Checks AEAD using nonce(CID sequence 1, PN 0)	
1-RTT[1]: DCID=S2, PATH_RESPONSE[Y],	
	ACK_MP[Seq=1, PN=0], ... -->

Figure 2: Example of new path establishment

In Figure Figure 2, the endpoints first exchange new available Connection IDs with the NEW_CONNECTION_ID frame. In this example the client provides one Connection ID (C1 with sequence number 1), and server provides two Connection IDs (S1 with sequence number 1, and S2 with sequence number 2).

Before the client opens a new path by sending an packet on that path with a PATH_CHALLENGE frame, it has to check. whether there is an unused Connection IDs available for each side. In this example the client chooses the Connection ID S2 as the Destination Connection ID in the new path.

If the client has used all the allocated CID, it is supposed to retire those that are not used anymore, and the server is supposed to provide replacements, as specified in [QUIC-TRANSPORT]. Usually it is desired to provide one more connection ID as currently in used, to allow for new paths or migration.

8.2. Path Closure

In this example the client detects the network environment change (client's 4G/Wi-Fi is turned off, Wi-Fi signal is fading to a threshold, or the quality of RTT or loss rate is becoming worse) and wants to close the initial path.

In Figure Figure 3 the server's 1-RTT packets use DCID C1, which has a sequence number of 1, for the first path; the client's 1-RTT packets use DCID S2, which has a sequence number of 2. For the second path, the server's 1-RTT packets use DCID C2, which has a sequence number of 2; the client's 1-RTT packets use CID S3, which has a sequence number of 3. Note that two paths use different packet number space.

Thee client initiates the path closure for the path with ID 1 by sending a packet with an PATH_ABANDON frame. When the server received the PATH_ABANDON frame, it also sends an PATH_ABANDON frame in the next packet. Afterwards the connection IDs in both directions can be retired using the RETIRE_CONNECTION_ID frame.

Client

Server

```

(client tells server to abandon a path)
1-RTT[X]: DCID=S2 PATH_ABANDON[path_id=1]->
      (server tells client to abandon a path)
<-1-RTT[Y]: DCID=C1 PATH_ABANDON[path_id=2], ACK_MP[Seq=2, PN=X]
(client abandons the path that it is using)
1-RTT[U]: DCID=S3 RETIRE_CONNECTION_ID[2], ACK_MP[Seq=1, PN=Y] ->
      (server abandons the path that it is using)
<- 1-RTT[V]: DCID=C2 RETIRE_CONNECTION_ID[1], ACK_MP[Seq=3, PN=U]

```

Figure 3: Example of closing a path (path id type=0x00)

9. Implementation Considerations

TDB

10. New Frames

All the new frames MUST only be sent in 1-RTT packet, and MUST NOT use other encryption levels.

If an endpoint receives multipath-specific frames from packets of other encryption levels, it MUST return MP_PROTOCOL_VIOLATION as a connection error and close the connection.

10.1. PATH_ABANDON Frame

The PATH_ABANDON frame informs the peer to abandon a path. More complex path management can be made possible with additional extensions (e.g., PATH_STATUS frame in [I-D.liu-multipath-quic]).

PATH_ABANDON frames are formatted as shown in Figure 4.

```

PATH_ABANDON Frame {
  Type (i) = TBD-03 (experiments use 0xbaba05),
  Path Identifier (...),
  Error Code (i),
  Reason Phrase Length (i),
  Reason Phrase (...),
}

```

Figure 4: PATH_ABANDON Frame Format

PATH_ABANDON frames contain the following fields:

Path Identifier: An identifier of the path, which is formatted as shown in Figure 5.

- * Identifier Type: Identifier Type field is set to indicate the type of path identifier.
 - Type 0: Refer to the connection identifier used by the sender of the control frame when sending data over the specified path. This method SHOULD be used if this connection identifier is non-zero length. This method MUST NOT be used if this connection identifier is zero-length.
 - Type 1: Refer to the connection identifier used by the receiver of the control frame when sending data over the specified path. This method MUST NOT be used if this connection identifier is zero-length.
 - Type 2: Refer to the path over which the control frame is sent or received.
- * Path Identifier Content: A variable-length integer specifying the path identifier. If Identifier Type is 2, the Path Identifier Content MUST be empty.

```
Path Identifier {  
    Identifier Type (i) = 0x00..0x02,  
    [Path Identifier Content (i)],  
}
```

Figure 5: Path Identifier Format

Note: If the receiver of the PATH_ABANDON frame is using non-zero length Connection ID on that path, endpoint SHOULD use type 0x00 for path identifier in the control frame. If the receiver of the PATH_ABANDON frame is using zero-length Connection ID, but the peer is using non-zero length Connection ID on that path, endpoints SHOULD use type 0x01 for path identifier. If both endpoints are using 0-length Connection IDs on that path, endpoints SHOULD only use type 0x02 for path identifier.

Error Code: A variable-length integer that indicates the reason for abandoning this path.

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

Reason Phrase: Additional diagnostic information for the closure.

This can be zero length if the sender chooses not to give details beyond the Error Code value. This SHOULD be a UTF-8 encoded string [RFC3629], though the frame does not carry information, such as language tags, that would aid comprehension by any entity other than the one that created the text.

PATH_ABANDON frames SHOULD be acknowledged. If a packet containing a PATH_ABANDON frame is considered lost, the peer SHOULD repeat it.

If the Identifier Type is 0x00 or 0x01, PATH_ABANDON frames MAY be sent on any path, not only the path identified by the Path Identifier Content field. If the Identifier Type is 0x02, the PATH_ABANDON frame MUST only be sent on the path that is intended to be abandoned.

10.2. ACK_MP Frame

The ACK_MP frame (types TBD-00 and TBD-01; experiments use 0xbaba00..0xbaba01) is an extension of the ACK frame defined by [QUIC-TRANSPORT]. It is used to acknowledge packets that were sent on different paths when using multiple packet number spaces. If the frame type is TBD-01, ACK_MP frames also contain the sum of QUIC packets with associated ECN marks received on the connection up to this point.

ACK_MP frame is formatted as shown in Figure 6.

```
ACK_MP Frame {
  Type (i) = TBD-00..TBD-01 (experiments use 0xbaba00..0xbaba01),
  Packet Number Space Identifier (i),
  Largest Acknowledged (i),
  ACK Delay (i),
  ACK Range Count (i),
  First ACK Range (i),
  ACK Range (...) ...,
  [ECN Counts (...)],
}
```

Figure 6: ACK_MP Frame Format

Compared to the ACK frame specified in [QUIC-TRANSPORT], the following field is added.

Packet Number Space Identifier: An identifier of the path packet number space, which is the sequence number of Destination Connection ID of the 1-RTT packets which are acknowledged by the ACK_MP frame. If the endpoint receives 1-RTT packets with zero-length Connection ID, it SHOULD use Packet Number Space Identifier 0 in ACK_MP frames. If an endpoint receives a ACK_MP frame with a non-existing packet number space ID, it MUST treat this as a connection error of type MP_PROTOCOL_VIOLATION and close the connection.

When using a single packet number space, endhosts MUST NOT send ACK_MP frames. If an endhost receives an ACK_MP frame while a single packet number space was negotiated, it MUST treat this as a connection error of type MP_PROTOCOL_VIOLATION and close the connection.

11. Error Codes

Multi-path QUIC transport error codes are 62-bit unsigned integers following [QUIC-TRANSPORT].

This section lists the defined multipath QUIC transport error codes that can be used in a CONNECTION_CLOSE frame with a type of 0x1c. These errors apply to the entire connection.

MP_PROTOCOL_VIOLATION (experiments use 0xba01): An endpoint detected an error with protocol compliance that was not covered by more specific error codes.

12. IANA Considerations

This document defines a new transport parameter for the negotiation of enable multiple paths for QUIC, and two new frame types. The draft defines provisional values for experiments, but we expect IANA to allocate short values if the draft is approved.

The following entry in Table 3 should be added to the "QUIC Transport Parameters" registry under the "QUIC Protocol" heading.

Value	Parameter Name.	Specification
TBD (experiments use 0xbabf)	enable_multipath	Section 2

Table 3: Addition to QUIC Transport Parameters Entries

The following frame types defined in Table 4 should be added to the "QUIC Frame Types" registry under the "QUIC Protocol" heading.

Value	Frame Name	Specification
TBD-00 - TBD-01 (experiments use 0xbaba00-0xbaba01)	ACK_MP	Section 10.2
TBD-02 (experiments use 0xbaba05)	PATH_ABANDON	Section 10.1

Table 4: Addition to QUIC Frame Types Entries

The following transport error code defined in Table 5 should be added to the "QUIC Transport Error Codes" registry under the "QUIC Protocol" heading.

Value	Code	Description	Specification
TBD (experiments use 0xba01)	MP_PROTOCOL_VIOLATION	Multi-path protocol violation	Section 11

Table 5: Error Code for Multi-path QUIC

13. Security Considerations

TBD

14. Contributors

This document is a collaboration of authors that combines work from three proposals. Further contributors that were also involved one of the original proposals are:

- * Qing An
- * Zhenyu Li

15. Acknowledgments

TBD

16. References

16.1. 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/info/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/info/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/info/rfc2119>>.
- [RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <<https://www.rfc-editor.org/info/rfc3629>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.

16.2. Informative References

- [I-D.bonaventure-iccr-g-schedulers] Bonaventure, O., Piraux, M., Coninck, Q. D., Baerts, M., Paasch, C., and M. Amend, "Multipath schedulers", Work in Progress, Internet-Draft, draft-bonaventure-iccr-g-schedulers-02, 25 October 2021, <<https://www.ietf.org/archive/id/draft-bonaventure-iccr-g-schedulers-02.txt>>.
- [I-D.liu-multipath-quic] Liu, Y., Ma, Y., Huitema, C., An, Q., and Z. Li, "Multipath Extension for QUIC", Work in Progress, Internet-Draft, draft-liu-multipath-quic-04, 5 September 2021, <<https://www.ietf.org/archive/id/draft-liu-multipath-quic-04.txt>>.
- [OLIA] Khalili, R., Gast, N., Popovic, M., Upadhyay, U., and J.-Y. Le Boudec, "MPTCP is not pareto-optimal: performance issues and a possible solution", Proceedings of the 8th international conference on Emerging networking experiments and technologies, ACM , 2012.

[QUIC-Invariants]

Thomson, M., "Version-Independent Properties of QUIC",
RFC 8999, DOI 10.17487/RFC8999, May 2021,
<<https://www.rfc-editor.org/info/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/info/rfc9002>>.

[QUIC-Timestamp]

Huitema, C., "Quic Timestamps For Measuring One-Way
Delays", Work in Progress, Internet-Draft, draft-huitema-
quic-ts-06, 12 September 2021,
<<https://www.ietf.org/archive/id/draft-huitema-quic-ts-06.txt>>.

[RFC6356] Raiciu, C., Handley, M., and D. Wischik, "Coupled
Congestion Control for Multipath Transport Protocols",
RFC 6356, DOI 10.17487/RFC6356, October 2011,
<<https://www.rfc-editor.org/info/rfc6356>>.

Authors' Addresses

Yanmei Liu
Alibaba Inc.

Email: miaoji.lym@alibaba-inc.com

Yunfei Ma
Alibaba Inc.

Email: yunfei.ma@alibaba-inc.com

Quentin De Coninck
UCLouvain

Email: quentin.deconinck@uclouvain.be

Olivier Bonaventure
UCLouvain

Email: olivier.bonaventure@uclouvain.be

Christian Huitema
Private Octopus Inc.

Email: huitema@huitema.net

Mirja Kuehlewind (editor)
Ericsson

Email: mirja.kuehlewind@ericsson.com

QUIC
Internet-Draft
Intended status: Informational
Expires: 28 April 2022

C. Smith
Magic Leap, Inc.
I. Swett
Google LLC
25 October 2021

QUIC Extension for Reporting Packet Receive Timestamps
draft-smith-quic-receive-ts-00

Abstract

This document defines an extension to the QUIC transport protocol which supports reporting multiple packet receive timestamps using a new ACK_RECEIVE_TIMESTAMP frame type.

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/>.

Source for this draft and an issue tracker can be found at <https://github.com/wcsmith/draft-quic-receive-ts>.

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 28 April 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. Motivation	2
3. Conventions and Definitions	3
4. Extension Negotiation	3
4.1. Receive Timestamp Basis	4
5. ACK_RECEIVE_TIMESTAMP Frame	4
5.1. Timestamp Ranges	5
6. Discussion	6
6.1. Best-Effort Behavior	6
7. Security Considerations	6
8. IANA Considerations	7
9. References	7
9.1. Normative References	7
9.2. Informative References	7
Acknowledgments	7
Authors' Addresses	7

1. Introduction

The QUIC Transport Protocol [RFC9000] provides a secure, multiplexed connection for transmitting reliable streams of application data.

This document defines an extension to the QUIC transport protocol which supports reporting multiple packet receive timestamps using a new ACK_RECEIVE_TIMESTAMP frame type.

2. Motivation

QUIC congestion control ([RFC9002]) supports sampling round-trip time (RTT) by measuring the time from when a packet was sent to when it is acknowledged. However, more precise delay signals measured via packet receive timestamps have the potential to improve the accuracy of network bandwidth measurements and the effectiveness of congestion control, especially for latency-critical applications such as real-time video conferencing or game streaming.

Numerous existing algorithms and techniques leverage receive timestamps to improve transport performance. Examples include:

- * The WebRTC congestion control algorithm described in [I-D.ietf-rmcat-gcc] uses the difference between packet inter-departure and packet inter-arrival times as the input to its delay-based controller.
- * The pathChirp ([RRBNC]) technique estimates available bandwidth by measuring inter-arrival time of multiple packets.

Notably, these techniques require receive timestamps for more than one packet per round-trip in order to best measure the network.

3. 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.

4. Extension Negotiation

The use of the ACK_RECEIVE_TIMESTAMP frame is negotiated using the following two transport parameters (Section 7.2 of [RFC9000]):

`max_receive_timestamps_per_ack` (TBD): A variable-length integer indicating that the sending endpoint would like to receive ACK_RECEIVE_TIMESTAMP frames from the peer containing no more than the given maximum number of receive timestamps.

Upon receiving this parameter with a non-zero value, the peer SHOULD send ACK_RECEIVE_TIMESTAMP frames instead of ACK frames if new receive timestamp information is available. The peer MAY still send regular ACK frames (e.g. if no timing information is available), in which case the endpoint MUST still support processing regular ACK frames as defined by Section 19.3 of [RFC9000].

Each ACK_RECEIVE_TIMESTAMP frame sent MUST NOT contain more than the specified maximum number of receive timestamps, but MAY contain fewer or none.

`receive_timestamps_exponent` (TBD): A variable-length integer

indicating the exponent to be used when encoding and decoding timestamp delta fields in ACK_RECEIVE_TIMESTAMP frames sent by the peer (see Section 5.1). If this value is absent, a default value of 0 is assumed (indicating microsecond precision). Values above 20 are invalid.

4.1. Receive Timestamp Basis

Endpoints which send ACK_RECEIVE_TIMESTAMP frames must determine a value, `receive_timestamp_basis`, relative to which all receive timestamps for the session will be reported (see Section 5.1).

The value of `receive_timestamp_basis` MUST be less than the smallest receive timestamp reported, and MUST remain constant for the entire duration of the session.

TODO: Discuss (here or below) why receive timestamps are reported relative to the basis, rather than in absolute time to avoid clock synchronization between endpoints.

5. ACK_RECEIVE_TIMESTAMP Frame

Receivers send ACK_RECEIVE_TIMESTAMP (type=TBD) frames in place of-- and in the same manner as--regular ACK frames as described in Section 13.2 of [RFC9000]. However, ACK_RECEIVE_TIMESTAMP frames contain additional fields to report packet receive timestamps.

ACK_RECEIVE_TIMESTAMP frames are formatted as shown in Figure 1.

```
ACK_RECEIVE_TIMESTAMP Frame {
  Type (i) = TBD
  // Fields of the existing ACK (type=0x02) frame:
  Largest Acknowledged (i),
  ACK Delay (i),
  ACK Range Count (i),
  First ACK Range (i),
  ACK Range (...) ...,
  // Additional fields for ACK_RECEIVE_TIMESTAMP:
  Timestamp Range Count (i),
  Timestamp Ranges (...) ...,
}
```

Figure 1: ACK_RECEIVE_TIMESTAMP Frame Format

The fields Largest Acknowledged, ACK Delay, ACK Range Count, First ACK Range, and ACK Range are the same as for ACK (type=0x02) frames specified in Section 19.3 of [RFC9000].

ACK_RECEIVE_TIMESTAMP frames contain the following additional fields:

Timestamp Range Count: A variable-length integer specifying the number of Timestamp Range fields in the frame.

Timestamp Ranges: Ranges of receive timestamps for contiguous packets in descending packet number order; see Section 5.1.

5.1. Timestamp Ranges

Each Timestamp Range describes a series of contiguous packet receive timestamps in descending sequential packet number (and descending timestamp) order. Timestamp Ranges consist of a Gap indicating the largest packet number in the range, followed by a list of Timestamp Deltas describing the relative receive timestamps for each contiguous packet in the Timestamp Range (descending).

Note that reporting receive timestamps for packets received out of order is not supported. Specifically: for any packet number *P* for which a receive timestamp *T* is reported, all reports for packet numbers less than *P* must have timestamps less than or equal to *T*; and all reports for packet numbers greater than *P* must have timestamps greater than or equal to *T*.

Timestamp Ranges are structured as shown in Figure 2.

```
Timestamp Range {  
    Gap (i),  
    Timestamp Delta Count (i),  
    Timestamp Delta (i) ...,  
}
```

Figure 2: Timestamp Range Format

The fields that form each Timestamp Range are:

Gap: A variable-length integer indicating the largest packet number in the Timestamp Range as follows:

For the first Timestamp Range: Gap is the difference between (a) the Largest Acknowledged packet number in the frame and (b) the largest packet in the current (first) Timestamp Range.

For subsequent Timestamp Ranges: Gap is the difference between (a) the packet number two lower than the smallest packet number in the previous Timestamp Range and (b) the largest packet in the current Timestamp Range.

Timestamp Delta Count: A variable-length integer indicating the number of Timestamp Deltas in the current Timestamp Range.

The sum of Timestamp Delta Counts for all Timestamp Ranges in the frame MUST NOT exceed `max_receive_timestamps_per_ack` as specified in Section 4.

Timestamp Deltas: Variable-length integers encoding the receive timestamp for contiguous packets in the Timestamp Range in descending packet number order as follows:

For the first Timestamp Delta of the first Timestamp Range in the frame: the value is the difference between (a) the receive timestamp of the largest packet in the Timestamp Range (indicated by Gap) and (b) the session `receive_timestamp_basis` (see Section 4.1), decoded as described below.

For all other Timestamp Deltas: the value is the difference between (a) the receive timestamp specified by the previous Timestamp Delta and (b) the receive timestamp of the current packet in the Timestamp Range, decoded as described below.

All Timestamp Delta values are decoded by multiplying the value in the field by 2 to the power of the `receive_timestamps_exponent` transport parameter received by the sender of the `ACK_RECEIVE_TIMESTAMP` frame (see Section 4):

6. Discussion

6.1. Best-Effort Behavior

Receive timestamps are sent on a best-effort basis and endpoints MUST gracefully handle scenarios where receive timestamp information for sent packets is not received. Examples of such scenarios are:

- * The packet containing the `ACK_RECEIVE_TIMESTAMP` frame is lost.
- * The sender truncates the number of timestamps sent in order to (a) avoid sending more than `max_receive_timestamps_per_ack` (Section 4); or (b) fit the ACK frame into a packet.

7. Security Considerations

TODO Security

8. IANA Considerations

This document has no IANA actions.

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>>.

9.2. Informative References

- [I-D.ietf-rmcat-gcc] Holmer, S., Lundin, H., Carlucci, G., Cicco, L. D., and S. Mascolo, "A Google Congestion Control Algorithm for Real-Time Communication", Work in Progress, Internet-Draft, draft-ietf-rmcat-gcc-02, 8 July 2016, <<https://datatracker.ietf.org/doc/html/draft-ietf-rmcat-gcc-02>>.
- [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>>.
- [RRBNC] Cottrel, R.V.R.R.B.R.N.J.a.L., "pathChirp: Efficient Available Bandwidth Estimation for Network Paths", 2003.

Acknowledgments

TODO acknowledge.

Authors' Addresses

Connor Smith
Magic Leap, Inc.

Email: connorsmith.ietf@gmail.com

Ian Swett
Google LLC

Email: ianswett@google.com