QUIC
Internet-Draft
Intended status: Informational
Expires: 30 January 2021

M. Duke
F5 Networks, Inc.

29 July 2020

Network Address Translation Support for QUIC
draft-duke-quic-natsupp-03

Abstract

Network Address Translators (NATs) are widely deployed to share scarce public IPv4 addresses among multiple end hosts. They overwrite IP addresses and ports in IP packets to do so. QUIC is a protocol on top of UDP that provides transport-like services. QUIC is better-behaved in the presence of NATs than older protocols, and existing UDP NATs should operate without incident if unmodified. QUIC offers additional features that may tempt NAT implementers as potential optimizations. However, in practice, leveraging these features will lead to new connection failure modes and security vulnerabilities.

Table of Contents

1. Introduction

Network Address Translators (NATs) are a widely deployed means of multiplexing multiple private IP addresses over scarce IPv4 public address space by replacing those addresses and using ports to distinguish those connections. The new address can also guarantee that packets move through a proxy throughout the life of a connection, so that the connection can continue with the required state at that proxy.

This document uses the colloquial term NAT to mean NAPT (section 2.2 of [RFC3022]), which overloads several IP addresses to one IP address or to an IP address pool, as commonly deployed in carrier-grade NATs or residential NATs.

QUIC [QUIC-TRANSPORT] is a protocol, operating over UDP, that provides many transport-like services to the application layer. Among these services is the mapping of multiple endpoint IP addresses to a single connection through use of a Connection ID (CID). Connection IDs are opaque byte fields that are expressed consistently across all QUIC versions [QUIC-INVARIANTS]. This feature may appear to present opportunities to optimize NAT port usage and simplify the work of the QUIC server. In fact, NAT behavior that relies on CID may instead cause connection failure when endpoints change Connection ID, and disable important protocol security features.

The remainder of this document explains how QUIC supports NATs better than other connection-oriented protocols, why NAT use of Connection ID might appear attractive, and how NAT use of CID can create serious problems for the endpoints. The conclusion of this document is that NATs should retain their existing 4-tuple-based operation and refrain from parsing or otherwise using QUIC connection IDs.

[RFC4787] contains some guidance on building NATs to interact constructively with a wide range of applications. This document extends the discussion to QUIC.

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. QUIC and NAT Rebinding

An explicit goal of QUIC is to be robust to NAT rebinding. When a connection is idle for a long time, the NAT may guess it has terminated and assign the client port to a new connection. As TCP defines a connection by its address and port 4-tuple, a TCP packet will not appear to belong to any existing connection at the receiver.

QUIC endpoints identify their connections using a CID that is encoded in every packet. If the client attempts to resume communication, the packet will be assigned a new source IP and/or port. Incoming packets from the server will be misrouted and dropped until the client sends a packet from its new address.

Therefore, QUIC connections can survive NAT rebindings as long as no routing function in the path is dependent on client IP address and port to deliver packets between server and NAT. Reducing the timeout on UDP NATs might be tempting in light of this property, but not all QUIC server deployments will be robust to rebinding.

4. The Lure of the Connection ID

There are a few reasons that CID-aware NATs could seemingly appear attractive.

4.1. Resource Conservation

NATs sometimes hit an operational limit where they exhaust available public IP addresses and ports, and must evict flows from their address/port mapping. CIDs offer a way to multiplex many connections over a single address and port.

However, QUIC endpoints may negotiate new connection IDs inside cryptographically protected packets, and begin using them at will. Imagine two clients behind a NAT that are sharing the same public IP address and port. The NAT is differentiating them using the incoming Connection ID. If one client secretly changes its connection ID, there will be no mapping for the NAT, and the connection will suddenly break.

While mid-connection failure in some cases may seem superior to rejecting QUIC outright, HTTP/3 over QUIC falls back to TCP. This is preferable to a connection suddenly black holing and timing out. Furthermore, wide deployment of NATs with this behavior would make it risky to change Connection IDs in the internet, which would thwart various important protocol properties.

It is possible, in principle, to encode the client's identity in a connection ID using [QUIC-LB] and explicit coordination with the NAT. However, QUIC-LB makes assumptions about endpoint mobility and common configuration in server infrastructure that are almost never valid in client/NAT architectures. Deploying such a system would include the administrative overhead while not solving the problem described in this section if the client changes networks.

Note that using connection IDs in this manner would anyway violate the best common practice to avoid "port overloading" as described in [RFC4787].

4.2. "Helping" with routing infrastructure issues

One problem in QUIC deployment is router and switch server infrastructures that direct traffic based on address-port 4-tuple rather than connection ID. The use of source IP address means that a NAT rebinding or address migration will deliver packets to the wrong server. For the reasons described above, routers and switches will not have access to negotiated CIDs. This is a particular problem for low-state load balancers, and a QUIC extension exists [QUIC-LB] to allow some server-load balancer coordination for routable CIDs.

A NAT at the front of this infrastructure might save the effort of converting all these devices by decoding routable connection IDs and rewriting the packet IP addresses to allow consistent routing by legacy devices.

Unfortunately, the change of IP address or port is an important signal to QUIC endpoints. It requires a review of path-dependent variables like congestion control parameters. It can also signify various attacks that mislead one endpoint about the best peer address for the connection (see section 9 of [QUIC-TRANSPORT]). The QUIC PATH_CHALLENGE and PATH_RESPONSE frames are intended to detect and mitigate these attacks and verify connectivity to the new address. This mechanism cannot work if the NAT is bleaching peer address changes.

For example, an attacker might copy a legitimate QUIC packet and change the source address to match its own. In the absence of a bleaching NAT, the receiving endpoint would interpret this as a potential NAT rebinding and use a PATH_CHALLENGE frame to prove that the peer endpoint is not truly at the new address, thus thwarting the attack. A bleaching NAT has no means of sending an encrypted PATH_CHALLENGE frame, so it might start redirecting all QUIC traffic to the attacker address and thus allow an observer to break the connection.

5. Filtering behavior

[RFC4787] describes possible packet filtering behaviors that relate to NATs. Though thes guidance there holds, a particularly unwise behavior is to admit a handful of UDP packets and then make a decision as to whether or not to filter it. QUIC applications are encouraged to fail over to TCP if early packets do not arrive at their destination. Admitting a few packets allows the QUIC endpoint to determine that the path accepts QUIC. Sudden drops afterwards will result in slow and costly timeouts before abandoning the connection.

6. QUIC Detection

Beyond the above difficulties, merely identifying that a UDP packet is part of a QUIC connection is not straightforward. Due to address migration, NATs cannot assume that QUIC version 1 application traffic is preceeded by a handshake on the path. The short header prepended to version 1 application traffic has few consistent codepoints that reliably identify it as QUIC. Moreover, the protocol is designed to be extensible. [QUIC-INVARIANTS] describes the small set of QUIC protocol properties that will remain stable across versions.

For these reasons, applying generalized UDP policies will prevent accidental breakage of QUIC features and mishandled non-QUIC UDP packets.

7. Security Considerations

This document proposes no change in behavior in the internet, so there are no new security implications. However, ignoring the recommendations here could prevent existing security mechanisms in QUIC from working properly.

8. IANA Considerations

There are no IANA requirements.

9. Informative References

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

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

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

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

[RFC3022] Srisuresh, P. and K. Egevang, "Traditional IP Network Address Translator (Traditional NAT)", RFC 3022, DOI 10.17487/RFC3022, January 2001, <https://www.rfc-editor.org/info/rfc3022>.

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

Appendix A. Acknowledgments

Thanks to Dmitri Tikhonov, who first recognized that certain NAT behaviors could create problems for QUIC.

Appendix B. Change Log

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

B.1. since draft-duke-quic-natsupp-02

* Added discussion of QUIC identification

B.2. since draft-duke-quic-natsupp-01

* Added brief discussion of impact of filtering.

* Added references to RFC 4787.

* Corrected normative reference to be informative.

B.3. since draft-duke-quic-natsupp-00

* Tightened NAT terminology

* Added additional clarfying examples

* Added warning against using QUIC-LB for NATs that front clients.

Author's Address

Martin Duke
F5 Networks, Inc.

Email: martin.h.duke@gmail.com

QUIC
Internet-Draft
Intended status: Experimental
Expires: 3 May 2021

M. Duke
F5 Networks, Inc.

30 October 2020

QUIC Version Aliasing
draft-duke-quic-version-aliasing-04

Abstract

The QUIC transport protocol [QUIC-TRANSPORT] preserves its future extensibility partly by specifying its version number. There will be a relatively small number of published version numbers for the foreseeable future. This document provides a method for clients and servers to negotiate the use of other version numbers in subsequent connections and encrypts Initial Packets using secret keys instead of standard ones. If a sizeable subset of QUIC connections use this mechanism, this should prevent middlebox ossification around the current set of published version numbers and the contents of QUIC Initial packets, as well as improving the protocol's privacy properties.

Table of Contents

1. Introduction

The QUIC version number is critical to future extensibility of the protocol. Past experience with other protocols, such as TLS1.3 [RFC8446], shows that middleboxes might attempt to enforce that QUIC packets use versions known at the time the middlebox was implemented. This has a chilling effect on deploying experimental and standard versions on the internet.

Each version of QUIC has a "salt" [QUIC-TLS] that is used to derive the keys used to encrypt Initial packets. As each salt is published in a standards document, any observer can decrypt these packets and inspect the contents, including a TLS Client Hello. A subsidiary mechanism like Encrypted SNI [ENCRYPTED-SNI] might protect some of the TLS fields inside a TLS Client Hello.

This document proposes "QUIC Version Aliasing," a standard way for servers to advertise the availability of other versions inside the cryptographic protection of a QUIC handshake. These versions are syntactically identical to the QUIC version in which the communication takes place, but use a different salt. In subsequent communications, the client uses the new version number and encrypts its Initial packets with a key derived from the provided salt. These version numbers and salts are unique to the client.

If a large subset of QUIC traffic adopts his technique, middleboxes will be unable to enforce particular version numbers or policy based on Client Hello contents without incurring unacceptable penalties on users. This would simultaneously protect the protocol against ossification and improve its privacy properties.

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.

A "standard version" is a QUIC version that would be advertised in a QUIC version negotiation and conforms to a specification. Any aliased version corresponds to a standard version in all its formats and behaviors, except for the version number field in long headers.

An "aliased version" is a version with a number generated in accordance with this document. Except for the version field in long headers, it conforms entirely to the specification of the standard version.

2. Protocol Overview

When they instantiate a connection, servers select an alternate 32-bit version number, and optionally an initial token extension, for the next connection at random and securely derive a salt and Packet Length Offset from those values using a repeatable process. They communicate this using a transport parameter extension including the version, initial token extension, salt, Packet Length Offset, and an expiration time for that value.

If a client next connects to that server within the indicated expiration time, it MAY use the provided version number and encrypt its Initial Packets using a key derived from the provided salt. It adds the Packet Length Offset to the true packet length when encoding it in the long header. If the server provided an Initial Token Extension, the client puts it in the Initial Packet token field. If there is another token the client wishes to include, it appends the Initial Token Extension to that token. The server can reconstruct the salt and Packet Length Offset from the requested version and token, and proceed with the connection normally.

The Packet Length Offset is provides a low-cost way for the server to verify it can derive a valid salt from the inputs without trial decryption. This has important security implications, as described in Section 7.2.

When generating a salt and Packet Length Offset, servers can choose between doing so randomly and storing the mapping, or using a cryptographic process to transform the aliased version number and token extension into the salt. The two options provide a simple tradeoff between computational complexity and storage requirements.

Note that clients and servers MUST implement [QUIC-VERSION-NEGOTIATION] to use this specification. Therefore, servers list supported versions in Version Negotiation Packets. Both clients and servers list supported versions in Version Negotiation Transport Parameters.

3. The Version Alias Transport Parameter

3.1. Version Number Generation

Servers MUST use a random process to generate version numbers. This version number MUST NOT correspond to a QUIC version the server advertises in QUIC Version Negotiation packets or transport parameters. Servers SHOULD also exclude version numbers used in known specifications or experiments to avoid confusion at clients, whether or not they have plans to support those specifications.

Servers MUST NOT use client-controlled information (e.g. the client IP address) in the random process, see Section 7.4.

Servers MUST NOT advertise these versions in QUIC Version Negotiation packets.

3.2. Initial Token Extension (ITE) Generation

Servers SHOULD generate an Initial Token Extension (ITE) to provide additional entropy in salt generation. Two clients that receive the same version number but different extensions will not be able to decode each other's Initial Packets.

Servers MAY choose any length that will allow client Initial Packets to fit within the minimum QUIC packet size of 1200 octets. A four-octet extension is RECOMMENDED. The ITE MUST appear to be random to observers.

If a server supports multiple standard versions, it MUST either encode the standard version of the current connection in the ITE or store it in a lookup table.

If the server chooses to encode the standard version, it MUST be cryptographically protected.

Encoded standard versions MUST be robust to false positives. That is, if decoded with a new key, the version encoding must return as invalid, rather than an incorrect value.

Alternatively, servers MAY store a mapping of unexpired aliased versions and ITEs to standard versions. This mapping SHOULD NOT create observable patterns, e.g. one plaintext bit indicates if the standard version is 1 or 2.

The server MUST be able to distinguish ITEs from Resumption and Retry tokens in incoming Initial Packets that contain an aliased version number. As the server controls the lengths and encoding of each, there are many ways to guarantee this.

3.3. Salt and Packet Length Offset Generation

The salt is an opaque 20-octet field. It is used to generate Initial connection keys using the process described in [QUIC-TLS].

The Packet Length Offset is a 64-bit unsigned integer with a maximum value of 2^62 - 1. Clients MUST ignore a transport parameter with a value that exceeds this limit.

To reduce header overhead, servers MAY consistently use a Packet Length Offset of zero if and only if it either (1) never sends Retry packets, or (2) can guarantee, through the use of persistent storage or other means, that it will never lose the cryptographic state required to generate the salt before the promised expiration time. Section 7.2 describes the implications if it uses zero without meeting these conditions.

Servers MUST either generate a random salt and Packet Length Offset and store a mapping of aliased version and ITE to salt and offset, or generate the salt and offset using a cryptographic method that uses the version number, ITE, and only server state that is persistent across connections.

If the latter, servers MUST implement a method that it can repeat deterministically at a later time to derive the salt and offset from the incoming version number and ITE. It MUST NOT use client controlled information other than the version number and ITE; for example, the client's IP address and port.

3.4. Expiration Time

Servers should select an expiration time in seconds, measured from the instant the transport parameter is first sent. This time SHOULD be less than the time until the server expects to support new QUIC versions, rotate the keys used to encode information in the version number, or rotate the keys used in salt generation.

Furthermore, the expiration time SHOULD be short enough to frustrate a salt polling attack ({salt-polling}})

Conversely, an extremely short expiration time will often force the client to use standard QUIC version numbers and salts.

3.5. Format

This document defines a new transport parameter extension for QUIC with identifier 0x5641. The contents of the value field are indicated below.

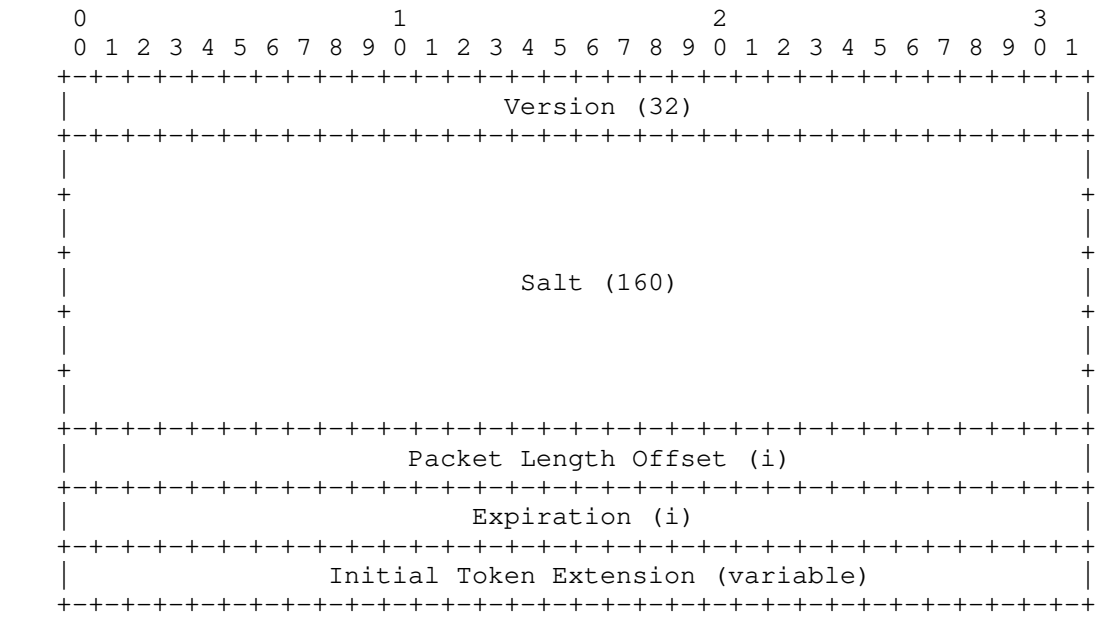

Figure 1: Version Alias Transport Parameter value

The definition of the fields is described above. Note that the "Expiration" field is in seconds, and its length is encoded using the Variable Length Integer encoding from Section 16 of [QUIC-TRANSPORT].

The Packet Length Offset is also encoded as a Variable Length Integer.

Clients can determine the length of the Initial Token Extension by subtracting known and encoded field lengths from the overall transport parameter length.

Note that servers that support version aliasing need not send the transport parameter on every connection. Therefore, a client MAY attempt to connect with an unexpired aliased version even if in its most recent connection it did not receive the transport parameter.

Clients MAY remember the value in this transport parameter for future connections. Servers MUST either store the contents of the transport parameter, or preserve the state to compute the full contents based on what the client provides.

3.6. Multiple Servers for One Domain

If multiple servers serve the same entity behind a load balancer, all such servers SHOULD either have a common configuration for encoding standard versions and computing salts, or share a common database of mappings. They MUST NOT generate version numbers that any of them would advertise in a Version Negotiation Packet or Transport Parameter.

4. Client Behavior

When a client receives the Version Alias Transport Parameter, it MAY cache the version number, ITE, salt, Packet Length Offset, and the expiration of these values. It MAY use the version number and ITE in a subsequent connection and compute the initial keys using the provided salt.

Clients MUST NOT advertise aliased versions in the Version Negotiation Transport Parameter unless they support a standard version with the same number. Including that number signals support for the standard version, not the aliased version.

Clients SHOULD NOT attempt to use the provided version number and salt after the provided Expiration time has elapsed.

Clients MAY decline to use the provided version number or salt in more than one connection. It SHOULD do so if its IP address has changed between two connection attempts. Using a consistent version number can link the client across connection attempts.

Clients MUST use the same standard version to format the Initial Packet as the standard version used in the connection that provided the aliased version.

If the server provided an ITE, the client MUST append it to any Initial Packet token it is including from a Retry packet or NEW_TOKEN frame, if it is using the associated aliased version. If there is no such token, it simply includes the ITE as the entire token.

The QUIC Token Length field MUST include the length of both any Retry or NEW_TOKEN token and the ITE.

The Length fields of all Initial, Handshake, and 0-RTT packets in the connection are set to the value described in [QUIC-TRANSPORT] plus the provided Packet Length Offset, modulo 2^62.

If the response to an Initial packet using the provided version is a Version Negotiation Packet, the client SHOULD cease attempting to use that version and salt to the server unless it later determines that the packet was the result of a version downgrade, see Section 7.1.

If a client receives an aliased version number that matches a standard version that the client supports, it SHOULD assume the server does not support the standard version and MUST use aliased version behaviors in any connection with the server using that version number.

If a client receives a Version Negotiation packet or Version Negotiation transport parameter advertising a version number the server previously sent as an aliased version, and the client verifies any Version Negotiation Packet is not a Version Downgrade attack (Section 7.1), it MUST discard the aliased version number, ITE, packet length offset, and salt and not use it in future connections.

5. Server Actions on Aliased Version Numbers

When a server receives an Initial Packet with an unsupported version number, it SHOULD send a Version Negotiation Packet if it is specifically configured not to generate that version number at random.

Otherwise, it extracts the ITE, if any, and either looks up the corresponding salt in its database or computes it using the technique originally used to derive the salt from the version number and ITE.

The server similarly obtains the Packet Length Offset and subtracts it from the provided Length field, modulo 2^62. If the resulting value is larger than the entire UDP datagram, the server discards the packet and SHOULD send a Version Negotiation Packet.

If the server supports multiple standard versions, it uses the standard version extracted by the ITE or stored in the mapping to parse the decrypted packet.

In all packets with long headers, the server uses the aliased version number and adds the Packet Length Offset to the length field.

In the extremely unlikely event that the Packet Length Offset resulted in a legal value but the salt is incorrect, the packet may fail authentication. If so, or the encoded standard version is not supported at the server, the server SHOULD send a Version Negotiation Packet.

To reduce linkability for the client, servers SHOULD provide a new Version Alias transport parameter, with a new version number, ITE, salt, and Packet Length Offset, each time a client connects. However, issuing version numbers to a client SHOULD be rate-limited to mitigate the salt polling attack Section 7.4.

6. Considerations for Retry Packets

QUIC Retry packets reduce the load on servers during periods of stress by forcing the client to prove it possesses the IP address before the server decrypts any Initial Packets or establishes any connection state. Version aliasing substantially complicates the process.

If a server has to send a Retry packet, the required format is ambiguous without understanding which standard version to use. If all supported standard versions use the same Retry format, it simply uses that format with the client-provided version number.

If the supported standard versions use different Retry formats, the server obtains the standard version via lookup or decoding and formats a Retry containing the aliased version number accordingly.

Servers generate the Retry Integrity Tag of a Retry Packet using the procedure in Section 5.8 of [QUIC-TLS]. However, for aliased versions, the secret key K uses the first 16 octets of the aliased salt instead of the key provided in the specification.

Clients MUST ignore Retry packets that contain a QUIC version other than the version it used in its Initial Packet.

Servers MUST NOT reply to a packet with an incorrect Length field in its long header with a Retry packet; it SHOULD reply with Version Negotiation as described above.

7. Security and Privacy Considerations

This document intends to improve the existing security and privacy properties of QUIC by dramatically improving the secrecy of QUIC Initial Packets. However, there are new attacks against this mechanism.

7.1. Version Downgrade

A countermeasure against version aliasing is the downgrade attack. Middleboxes may drop a packet containing a random version and imitate the server's failure to correctly process it. Clients and servers are required to implement [QUIC-VERSION-NEGOTIATION] to detect downgrades.

Note that downgrade detection only works after receiving a response from the server. If a client immediately responds to a Version Negotiation Packet with an Initial Packet with a standard version number, it will have exposed its request in a format readable to observers before it discovers if the Version Negotiation Packet is authentic. A client SHOULD wait for an interval to see if a valid response comes from the server before assuming the version negotiation is valid. The client MAY also alter its Initial Packet (e.g., its ALPN field) to sanitize sensitive information and obtain another aliased version before proceeding with its true request.

Servers that support version aliasing SHOULD be liberal about the Initial Packet content they receive, keeping the connection open long enough to deliver their transport parameters, to support this mechanism.

7.2. Retry Injection

QUIC Version 1 Retry packets are spoofable, as they follow a fixed format, are sent in plaintext, and the integrity protection uses a widely known key. As a result, QUIC Version 1 has verification mechanisms in subsequent packets of the connection to validate the origin of the Retry.

Version aliasing largely frustrates this attack. As the integrity check key is derived from the secret salt, packets from attackers will fail their integrity check and the client will ignore them.

The Packet Length Offset is important in this framework. Without this mechanism, servers would have to perform trial decryption to verify the client was using the correct salt. As this does not occur before sending Retry Packets, servers would not detect disagreement on the salt beforehand and would send a Retry packet signed with a different salt than the client expects. Therefore, a client that received a Retry packet with an invalid integrity check would not be able to distinguish between the following possibilities:

* a Retry packet corrupted in the network, which should be ignored;

* a Retry packet generated by an attacker, which should be ignored; or

* a Retry packet from a server that lost its cryptographic state, meaning that further communication with aliased versions is impossible and the client should revert to using a standard version.

The Packet Length Offset introduces sufficient entropy to make the third possibility exceedingly unlikely.

7.3. Increased Linkability

As each version number and ITE is unique to each client, if a client uses one twice, those two connections are extremely likely to be from the same host. If the client has changed IP address, this is a significant increase in linkability relative to QUIC with a standard version numbers.

7.4. Salt Polling Attack

Observers that wish to decode Initial Packets might open a large number of connections to the server in an effort to obtain part of the mapping of version numbers and ITEs to salts for a server. While storage-intensive, this attack could increase the probability that at least some version-aliased connections are observable. There are three mitigations servers can execute against this attack:

* use a longer ITE to increase the entropy of the salt,

* rate-limit transport parameters sent to a particular client, and/or

* set a low expiration time to reduce the lifetime of the attacker's database.

Segmenting the version number space based on client information, i.e. using only a subset of version numbers for a certain IP address range, would significantly amplify an attack. Observers will generally be on the path to the client and be able to mimic having an identical IP address. Segmentation in this way would dramatically reduce the search space for attackers. Thus, servers are prohibited from using this mechanism.

7.5. Increased Processing of Garbage UDP Packets

As QUIC shares the UDP protocol number with other UDP applications, in some deployments it may be possible for traffic intended for other UDP applications to arrive at a QUIC server endpoint. When servers support a finite set of version numbers, a valid version number field is a strong indicator the packet is, in fact, QUIC. If the version number is invalid, a QUIC Version Negotiation is a low-cost response that triggers very early in packet processing.

However, a server that provides version aliasing is prepared to accept almost any version number. As a result, many more sufficiently sized UDP payloads with the first bit set to '1' are potential QUIC Initial Packets that require generation of a salt and Packet Length Offset.

Note that a nonzero Packet Length Offset will allow the server to drop all but approximately 1 in every 2^49 packets, so trial decryption is unnecessary.

While not a more potent attack then simply sending valid Initial Packets, servers may have to provision additional resources to address this possibility.

7.6. Increased Retry Overhead

This document requires two small cryptographic operations to build a Retry packet instead of one, placing more load on servers when already under load.

7.7. Request Forgery Attacks

Section 21.4 of [QUIC-TRANSPORT] describes the request forgery attack, where a QUIC endpoint can cause its peer to deliver packets to a victim with specific content.

Version aliasing allows the server to specify the contents of the version field and part of the token field in Initial packets sent by the client, potentially increasing the potency of this attack.

8. IANA Considerations

This draft chooses a transport parameter (0x5641) to minimize the risk of collision. IANA should assign a permanent value from the QUIC Transport Parameter Registry.

9. References

9.1. Normative References

[QUIC-TLS] Thomson, M., Ed. and S. Turner, Ed., "Using Transport Layer Security (TLS) to Secure QUIC", Work in Progress, Internet-Draft, draft-ietf-quic-tls-latest, <https://tools.ietf.org/html/draft-ietf-quic-tls-latest>.

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

[QUIC-VERSION-NEGOTIATION] Schinazi, D., Ed. and E. Rescorla, Ed., "Compatible Version Negotiation for QUIC", Work in Progress, Internet-Draft, draft-ietf-quic-version-negotiation-latest, <https://tools.ietf.org/html/draft-ietf-quic-version-negotiation-latest>.

9.2. Informative References

[ENCRYPTED-SNI] Rescorla, E., Ed., Oku, K., Ed., Sullivan, N., Ed., and C.A. Wood, Ed., "Encrypted Server Name Indication for TLS 1.3", Work in Progress, Internet-Draft, draft-ietf-tls-esni-latest, <https://tools.ietf.org/html/draft-ietf-tls-esni-latest>.

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

[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>.

Appendix A. Acknowledgments

Marten Seemann was the original creator of the version aliasing approach.

Appendix B. Change Log

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

B.1. since draft-duke-quic-version-aliasing-03

* Discussed request forgery attacks

B.2. since draft-duke-quic-version-aliasing-02

* Specified 0RTT status of the transport parameter

B.3. since draft-duke-quic-version-aliasing-01

* Fixed all references to "seed" where I meant "salt."

* Added the Packet Length Offset, which eliminates Retry Injection Attacks

B.4. since draft-duke-quic-version-aliasing-00

* Added "Initial Token Extensions" to increase salt entropy and make salt polling attacks impractical.

* Allowed servers to store a mapping of version number and ITE to salt instead.

* Made standard version encoding mandatory. This dramatically simplifies the new Retry logic and changes the security model.

* Added references to Version Negotiation Transport Parameters.

* Extensive readability edit.

Author's Address

Martin Duke
F5 Networks, Inc.

Email: martin.h.duke@gmail.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: 6 May 2021

M. Kuehlewind
Ericsson
B. Trammell
Google

2 November 2020

Applicability of the QUIC Transport Protocol
draft-ietf-quic-applicability-08

Abstract

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

Table of Contents

1. Introduction

QUIC [QUIC] is a new transport protocol providing a number of advanced features. While initially designed for the HTTP use case, like most transports it is intended for use with a much wider variety of applications. QUIC is encapsulated in UDP. The version of QUIC that is currently under development will integrate TLS 1.3 [TLS13] to encrypt all payload data and most control information. HTTP operating over QUIC is known as HTTP/3.

This document provides guidance for application developers that want to use the QUIC protocol without implementing it on their own. This includes general guidance for applications operating over HTTP/3 or directly over QUIC. For specific guidance on how to integrate HTTP/3 with QUIC, see [QUIC-HTTP].

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

1.1. Notational Conventions

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

2. The Necessity of Fallback

QUIC uses UDP as a substrate for userspace implementation and port numbers for NAT and middlebox traversal. While there is no evidence of widespread, systematic disadvantage of UDP traffic compared to TCP in the Internet [Edeline16], somewhere between three [Trammell16] and five [Swett16] percent of networks simply block UDP traffic. All applications running on top of QUIC must therefore either be prepared to accept connectivity failure on such networks, or be engineered to fall back to some other transport protocol. This fallback SHOULD provide TLS 1.3 or equivalent cryptographic protection, if available, in order to keep fallback from being exploited as a downgrade attack. In the case of HTTP, this fallback is TLS 1.3 over TCP.

These applications must operate, perhaps with impaired functionality, in the absence of features provided by QUIC not present in the fallback protocol. For fallback to TLS over TCP, the most obvious difference is that TCP does not provide stream multiplexing and therefore stream multiplexing would need to be implemented in the application layer if needed.

Further, TCP implementations and network paths often do not support the Fast Open option, which is analogous to 0-RTT session resumption. Even if Fast Open successfully operates end-to-end, it is limited to a single packet of payload, unlike QUIC 0-RTT.

Note that there is some evidence of middleboxes blocking SYN data even if TFO was successfully negotiated (see [PaaschNanog]).

Any fallback mechanism is likely to impose a degradation of performance; however, fallback MUST not silently violate the application's expectation of confidentiality or integrity of its payload data.

Moreover, while encryption (in this case TLS) is inseparably integrated with QUIC, TLS negotiation over TCP can be blocked. In case it is RECOMMENDED to abort the connection, allowing the application to present a suitable prompt to the user that secure communication is unavailable.

3. Zero RTT

QUIC provides for 0-RTT connection establishment. This presents opportunities and challenges for applications using QUIC.

3.1. Thinking in Zero RTT

A transport protocol that provides 0-RTT connection establishment to recently contacted servers is qualitatively different than one that does not from the point of view of the application using it. Relative trade-offs between the cost of closing and reopening a connection and trying to keep it open are different; see Section 3.3.

Applications must be slightly rethought in order to make best use of 0-RTT resumption. Most importantly, application operations must be divided into idempotent and non-idempotent operations, as only idempotent operations may appear in 0-RTT packets. This implies that the interface between the application and transport layer exposes idempotence either explicitly or implicitly.

3.2. Here There Be Dragons

Retransmission or (malicious) replay of data contained in 0-RTT resumption packets could cause the server side to receive two copies of the same data. This is further described in [HTTP-RETRY]. Data sent during 0-RTT resumption also cannot benefit from perfect forward secrecy (PFS).

Data in the first flight sent by the client in a connection established with 0-RTT MUST be idempotent (as specified in section 2.1 in [QUIC-TLS]). Applications MUST be designed, and their data MUST be framed, such that multiple reception of idempotent data is recognized as such by the receiver. Applications that cannot treat data that may appear in a 0-RTT connection establishment as idempotent MUST NOT use 0-RTT establishment. For these reason the QUIC transport SHOULD provide some or all of the following interfaces to applications:

* indicate if 0-RTT support is in general desired, which implies that lack of PFS is acceptable for some data;

* an indication when 0RTT data for both egress and ingress, so that both sender and receiver understand the properties of the communication channel when the data is sent; and/or

* whether rejected 0-RTT data should be retransmitted or withdrawn.

Some TLS implementations may offer replay protection, which may mitigate some of these issues.

3.3. Session resumption versus Keep-alive

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

By using a Connection ID, QUIC is designed to be robust to NAT address rebinding after a timeout. However, some QUIC connections may not be robust to rebinding because the routing infrastructure (in particular, load balancers) uses the address/port four-tuple to direct traffic. Furthermore, middleboxes with functions other than address translation may still affect the path. In particular, firewalls will often not admit server traffic for which it has not kept state for corresponding packets from the client.

A QUIC application has three strategies to deal with this issue by adjusting idle periods (noting that idle periods and the network idle timeout is distinct from the connection idle timeout, defined as the minimum of the idle timeout parameter in Section 10.1 of [QUIC]):

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

* Ensure there are no long idle periods.

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

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

Either the server or the client in a QUIC application can send PING frames as keep-alives, to prevent the connection and any on-path state from timing out. Recommendations for the use of keep-alives are application specific, mainly depending on the latency requirements and message frequency of the application. In this case, the application mapping must specify whether the client or server is responsible for keeping the application alive. While [Hatonen10] suggests that 30 seconds might be a suitable value for the public Internet when a NAT is on path, larger values are preferable if the deployment can consistently survive NAT rebinding, or is known to be in a controlled environments like e.g. data centres in order to lower network and computational load. Sending PING frames more frequently than every 30 seconds over long idle periods may result in excessive unproductive traffic in some situations, and to unacceptable power usage for power-constrained (mobile) devices. Additionally, time-outs shorter than 30 seconds can make it harder to handle trasient network interruptions, such as VM migration or coverage loss during mobilty.

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

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

4. Use of Streams

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

Streams can be unidirectional or bidirectional, and a stream may be initiated either by client or server. Only the initiator of a unidirectional stream can send data on it. Due to offset encoding limitations, a stream can carry a maximum of 2^62-1 bytes in each direction. In the presently unlikely event that this limit is reached by an application, the stream can simply be closed and replaced with a new one.

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

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

Mapping of application data to streams is application-specific and described for HTTP/3 in [QUIC-HTTP]. In general, data that can be processed independently, and therefore would suffer from head of line blocking if forced to be received in order, should be transmitted over separate streams. If the application requires certain data to be received in order, that data should be sent on the same stream. If there is a logical grouping of data chunks or messages, streams can be reused, or a new stream can be opened for each chunk/message. If one message is mapped to a single stream, resetting the stream to expire an unacknowledged message can be used to emulate partial reliability on a message basis. If a QUIC receiver has maximum allowed concurrent streams open and the sender on the other end indicates that more streams are needed, it doesn’t automatically lead to an increase of the maximum number of streams by the receiver. Therefore it can be valuable to expose maximum number of allowed, currently open and currently used streams to the application to make the mapping of data to streams dependent on this information.

While a QUIC implementation must necessarily provide a way for an application to send data on separate streams, it does not necessarily expose stream identifiers to the application (see e.g. [QUIC-HTTP] section 6) either at the sender or receiver end, so applications should not assume access to these identifiers.

4.1. Stream versus Flow Multiplexing

Streams are meaningful only to the application; since stream information is carried inside QUIC's encryption boundary, no information about the stream(s) whose frames are carried by a given packet is visible to the network. Therefore stream multiplexing is not intended to be used for differentiating streams in terms of network treatment. Application traffic requiring different network treatment SHOULD therefore be carried over different five-tuples (i.e. multiple QUIC connections). Given QUIC's ability to send application data in the first RTT of a connection (if a previous connection to the same host has been successfully established to provide the respective credentials), the cost of establishing another connection is extremely low.

4.2. Prioritization

Stream prioritization is not exposed to either the network or the receiver. Prioritization is managed by the sender, and the QUIC transport should provide an interface for applications to prioritize streams [QUIC]. Further applications can implement their own prioritization scheme on top of QUIC: an application protocol that runs on top of QUIC can define explicit messages for signaling priority, such as those defined for HTTP/2; it can define rules that allow an endpoint to determine priority based on context; or it can provide a higher level interface and leave the determination to the application on top.

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

4.3. Flow Control Deadlocks

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

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

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

A common flow control implementation technique is for a receiver to extend credit to the sender as a the data consumer reads data. In this setting, a length-prefixed message format makes it easier for the data consumer to leave data unread in the receiver's buffers and thereby withhold flow control credit. If flow control limits prevent the remainder of a message from being sent, a deadlock will result. A length prefix might also enable the detection of this sort of deadlock. Where protocols have messages that might be processed as a single unit, reserving flow control credit for the entire message atomically ensures that this style of deadlock is less likely.

A data consumer can read all data as it becomes available to cause the receiver to extend flow control credit to the sender and reduce the chances of a deadlock. However, releasing flow control credit might mean that the data consumer might need other means for holding a peer accountable for the state it keeps for partially processed messages.

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

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

5. Packetization and Latency

QUIC provides an interface that provides multiple streams to the application; however, the application usually cannot control how data transmitted over one stream is mapped into frames or how those frames are bundled into packets. By default, QUIC will try to maximally pack packets with one or more stream data frames to minimize bandwidth consumption and computational costs (see section 13 of [QUIC]). If there is not enough data available to fill a packet, QUIC may even wait for a short time, to optimize bandwidth efficiency instead of latency. This delay can either be pre-configured or dynamically adjusted based on the observed sending pattern of the application. If the application requires low latency, with only small chunks of data to send, it may be valuable to indicate to QUIC that all data should be send out immediately. Alternatively, if the application expects to use a specific sending pattern, it can also provide a suggested delay to QUIC for how long to wait before bundle frames into a packet.

Similarly, an appliaction has usually no control about the length of a QUIC packet on the wire. However, QUIC provides the ability to add a padding frame to impact the packet size. This is mainly used by QUIC itself in the first packet in order to ensure that the path is capable of transferring packets of at least a certain size. Additionally, a QUIC implementation can expose an application layer interface to specify a certain packet size. This can either be used by the application to force certian packet sizes in specific use cases/networks, or ensure that all packets are equally sized to conceal potential leakage of application layer information when the data sent by the application are not greedy. Note the initial packet must have a minimum size of 1200 bytes according to the QUIC specification. A receiver of a smaller initial packet may reject this packet in order to avoid amplification attacks.

6. Port Selection and Application Endpoint Discovery

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

As QUIC is a general purpose transport protocol, there are no requirements that servers use a particular UDP port for QUIC in general. For applications with a fallback to TCP which do not

already have an alternate mapping to UDP, the registration (if necessary) and use of the UDP port number corresponding to the TCP port already registered for the application is RECOMMENDED. For example, the default port for HTTP/3 [QUIC-HTTP] is UDP port 443, analogous to HTTP/1.1 or HTTP/2 over TLS over TCP.

Applications SHOULD define an alternate endpoint discovery mechanism to allow the usage of ports other than the default. For example, HTTP/3 ([QUIC-HTTP] sections 3.2 and 3.3) specifies the use of ALPN [RFC7301] for service discovery which allows the server to use and announce a different port number. Note that HTTP/3's ALPN token ("h3") identifies not only the version of the application protocol, but also the binding to QUIC as well as the version of QUIC itself; this approach allows unambiguous agreement between the endpoints on the protocol stack in use.

Note that given the prevalence of the assumption in network management practice that a port number maps unambiguously to an application, the use of ports that cannot easily be mapped to a registered service name may lead to blocking or other interference by network elements such as firewalls that rely on the port number for application identification.

7. Connection Migration

QUIC supports connection migration by the client. If a lower-layer address changes, a QUIC endpoint can still associate packets with an existing connection based on the Connection ID (see also Section 9) in the QUIC header, if present. This supports cases where address information changes, such as NAT rebinding, intentional change of the local interface, or based on an indication in the handshake of the server for a preferred address to be used. As such if the client is known or likely to sit behind a NAT, use of a connection ID for the server is strongly recommended. A non-empty connection ID for the server is also strongly recommended when migration is supported.

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

As long as the new path is not validated only probing packets can be sent. However, the probing packets can be used measure path characteristics as input for the switching decision or the congestion controller on the new path.

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

8. Connection closure

QUIC connections are closed either by expiration of an idle timeout, as determined by transport parameters, or by an explicit indication of the application that a connection should be closed (immediate close). While data could still be received after the immediate close has been initiated by one endpoint (for a limited time period), the expectation is that an immediate close was negotiated at the application layer and therefore no additional data is expected from both sides.

An immidate close will emit an CONNECTION_CLOSE frame. This frames has two sets of types: one for QUIC internal problems that might lead to connection closure, and one for closures initiated by the application. An application using QUIC can define application-specific error codes, e.g. see [QUIC-HTTP] section 8.1. In the case of a grateful shut-down initiated by the application after application layer negotiation, a NO_ERROR code is expected. Further, the CONNECTION_CLOSE frame provides an optional reason field, that can be used to append human-readable information to an error code. Note that QUIC RESET_STREAM and STOP_SENDING frames provide similar capablities. Usually application error codes are defined to be applicabile to all three frames.

Alternatively, a QUIC connection can be silently closed by each endpoint separately after an idle timeout. If enabled as indicated by a transport parameter in the handshake, the idle timeout is announced for each endpoint during connection establishment and the effective value for this connection is the minimum of the two advertised values. An application therefore should be able to configure its own maximum value as well as have access to the computed minimum value for this connection. An application may adjust the maximum idle timeout based on the number of open or expected connections as shorter timeout values may free-up memory more quickly. If an application desires to keep the connection open for longer than the announced timeout, it can send keep-alives messages, or a QUIC implementation may provide an option to defer the time-out to avoid unnecessary load, as specified in Section 10.2.2 of [QUIC]. See Section 3.3 for further guidance on keep-alives.

9. Information exposure and the Connection ID

QUIC exposes some information to the network in the unencrypted part of the header, either before the encryption context is established, because the information is intended to be used by the network. QUIC has a long header that is used during connection establishment and for other control processes, and a short header that may be used for data transmission in an established connection. While the long header always exposes some information (such as the version and Connection IDs), the short header exposes at most only a single Connection ID.

Note that the Connection ID in the short header may be omitted. This is a per-connection configuration option; if the Connection ID is not present, then the peer omitting the connection ID needs to use the same local address for the lifetime of the connection and connection migration is not supported for that direction of the connection.

9.1. Server-Generated Connection ID

QUIC supports a server-generated Connection ID, transmitted to the client during connection establishment (see Section 6.1 of [QUIC]). Servers behind load balancers may need to change the Connection ID during the handshake, encoding the identity of the server or information about its load balancing pool, in order to support stateless load balancing. Once the server generates a Connection ID that encodes its identity, every CDN load balancer would be able to forward the packets to that server without retaining connection state.

9.2. Mitigating Timing Linkability with Connection ID Migration

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

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

The most efficient mitigation for these attacks is operational, either by using a load balancing architecture that loads more flows onto a single server-side address, by coordinating the timing of migrations to attempt to increase the number of simultaneous migrations at a given time, or through other means.

9.3. Using Server Retry for Redirection

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

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

10. Use of Versions and Cryptographic Handshake

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

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

11. Enabling New Versions

QUIC provides integrity protection for its version negotiation process. This process assumes that the set of versions that a server supports is fixed. This complicates the process for deploying new QUIC versions or disabling old versions when servers operate in clusters.

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

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

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

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

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

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

The third stage completes the process by enabling validation of the negotiation version as though the new version were disabled.

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

12. IANA Considerations

This document has no actions for IANA; however, note that Section 6 recommends that application bindings to QUIC for applications using TCP register UDP ports analogous to their existing TCP registrations.

13. Security Considerations

See the security considerations in [QUIC] and [QUIC-TLS]; the security considerations for the underlying transport protocol are relevant for applications using QUIC, as well.

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

14. Contributors

Igor Lubashev contributed text to Section 9 on server-selected Connection IDs.

15. Acknowledgments

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

16. References

16.1. Normative References

[QUIC] Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", Work in Progress, Internet-Draft, draft-ietf-quic-transport-32, 20 October 2020, <http://www.ietf.org/internet-drafts/draft-ietf-quic-transport-32.txt>.

[QUIC-INVARIANTS] Thomson, M., "Version-Independent Properties of QUIC", Work in Progress, Internet-Draft, draft-ietf-quic-invariants-11, 24 September 2020, <http://www.ietf.org/internet-drafts/draft-ietf-quic-invariants-11.txt>.

[QUIC-TLS] Thomson, M. and S. Turner, "Using TLS to Secure QUIC", Work in Progress, Internet-Draft, draft-ietf-quic-tls-32, 20 October 2020, <http://www.ietf.org/internet-drafts/draft-ietf-quic-tls-32.txt>.

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

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

[TLS13] Thomson, M. and S. Turner, "Using TLS to Secure QUIC", Work in Progress, Internet-Draft, draft-ietf-quic-tls-32, 20 October 2020, <http://www.ietf.org/internet-drafts/draft-ietf-quic-tls-32.txt>.

16.2. Informative References

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

[Hatonen10]
Hatonen, S., Nyrhinen, A., Eggert, L., Strowes, S., Sarolahti, P., and M. Kojo, "An experimental study of home gateway characteristics (Proc. ACM IMC 2010)", October 2010.

[HTTP-RETRY]
Nottingham, M., "Retrying HTTP Requests", Work in Progress, Internet-Draft, draft-nottingham-httpbis-retry-01, 1 February 2017, <http://www.ietf.org/internet-drafts/draft-nottingham-httpbis-retry-01.txt>.

[I-D.nottingham-httpbis-retry]
Nottingham, M., "Retrying HTTP Requests", Work in Progress, Internet-Draft, draft-nottingham-httpbis-retry-01, 1 February 2017, <http://www.ietf.org/internet-drafts/draft-nottingham-httpbis-retry-01.txt>.

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

[QUIC-HTTP]
Bishop, M., "Hypertext Transfer Protocol Version 3 (HTTP/3)", Work in Progress, Internet-Draft, draft-ietf-quic-http-32, 20 October 2020, <http://www.ietf.org/internet-drafts/draft-ietf-quic-http-32.txt>.

[QUIC-LB] Duke, M. and N. Banks, "QUIC-LB: Generating Routable QUIC Connection IDs", Work in Progress, Internet-Draft, draft-ietf-quic-load-balancers-05, 30 October 2020, <http://www.ietf.org/internet-drafts/draft-ietf-quic-load-balancers-05.txt>.

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

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

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

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

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

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

Authors' Addresses

Mirja Kuehlewind
Ericsson

Email: mirja.kuehlewind@ericsson.com

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

Email: ietf@trammell.ch

QUIC
Internet-Draft
Intended status: Standards Track
Expires: 3 May 2021

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

30 October 2020

QUIC-LB: Generating Routable QUIC Connection IDs
draft-ietf-quic-load-balancers-05

Abstract

QUIC connection IDs allow continuation of connections across address/port 4-tuple changes, and can store routing information for stateless or low-state load balancers. They also can prevent linkability of connections across deliberate address migration through the use of protected communications between client and server. This creates issues for load-balancing intermediaries. This specification standardizes methods for encoding routing information given a small set of configuration parameters. This framework also enables offload of other QUIC functions to trusted intermediaries, given the explicit cooperation of the QUIC server.

Table of Contents

1. Introduction

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

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

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

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

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

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

1.1. Terminology

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

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

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

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

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

2. Protocol Objectives

2.1. Simplicity

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

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

2.2. Security

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

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

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

3. First CID octet

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

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

3.1. Config Rotation

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

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

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

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

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

It also possible to use these bits for more long-lived distinction of different configurations, but this has privacy implications (see Section 10.3).

3.2. Configuration Failover

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

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

3.3. Length Self-Description

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

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

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

3.4. Format

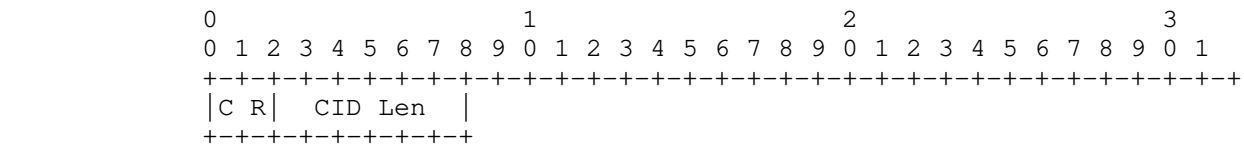

Figure 1: First Octet Format

The first octet has the following fields:

CR: Config Rotation bits.

CID Len: Length Self-Description (if applicable). Encodes the length of the Connection ID following the First Octet.

4. Routing Algorithms

In QUIC-LB, load balancers do not generate individual connection IDs to 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.

As clients sometimes generate the DCIDs in long headers, these might not conform to the expectations of the routing algorithm. These are called "non-compliant DCIDs":

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

* The config rotation bits (Section 3.1) may not correspond to an active configuration.

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

Load balancers MUST forward packets with long headers and non-compliant DCIDs to an active server using an algorithm of its own choosing. It need not coordinate this algorithm with the servers. The algorithm SHOULD be deterministic over short time scales so that related packets go to the same server. The design of this algorithm SHOULD consider the version-invariant properties of QUIC described in [QUIC-INVARIANTS] to maximize its robustness to future versions of QUIC. For example, a non-compliant DCID might be converted 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. See also Section 9.

As a partial exception to the above, load balancers MAY drop packets with long headers and non-compliant DCIDs if and only if it knows that the encoded QUIC version does not allow a non-compliant DCID in a packet with that signature. For example, a load balancer can safely drop a QUIC version 1 Handshake packet with a non-compliant DCID. The prohibition against dropping packets with long headers remains for unknown QUIC versions.

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

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

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

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

The load balancer 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 cannot base their design on version-specific templates.

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.

This section describes three participants: the configuration agent, the load balancer, and the server.

4.1. Plaintext CID Algorithm

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

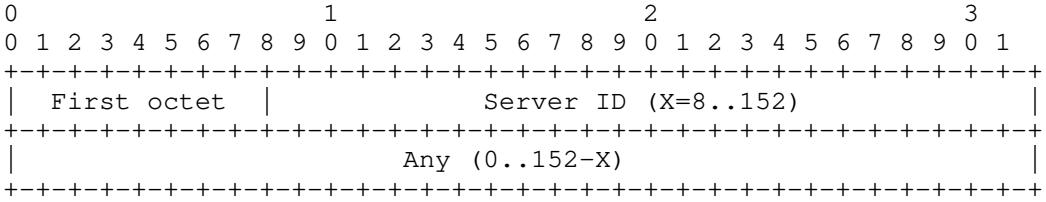

Figure 2: Plaintext CID Format

4.1.1. Configuration Agent Actions

The configuration agent selects a length for the server ID encoding. This length MUST have enough entropy to have a different code point for each server.

It also assigns a server ID to each server.

4.1.2. Load Balancer Actions

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

4.1.3. Server Actions

The server chooses a connection ID length. This MUST be at least one byte longer than the routing bytes.

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

4.2. Stream Cipher CID Algorithm

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

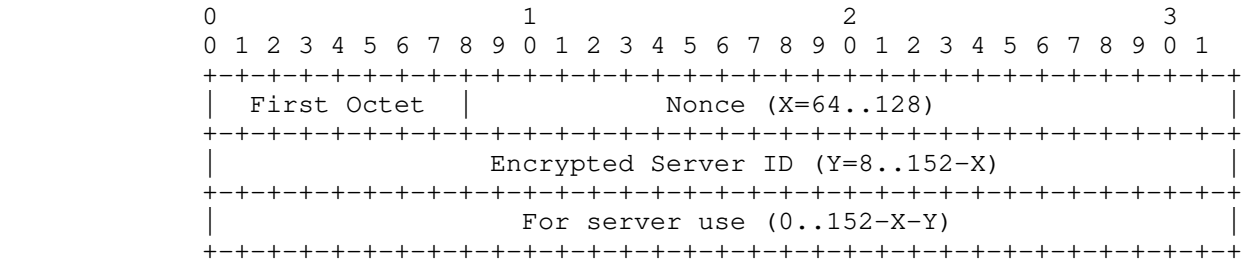

Figure 3: Stream Cipher CID Format

4.2.1. Configuration Agent Actions

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

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

4.2.2. Load Balancer Actions

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

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

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

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

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

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

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

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

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

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

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

4.2.3. Server Actions

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

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

The server encrypts the server ID using exactly the algorithm as described in Section 4.2.2, performing the three passes in reverse order.

4.3. Block Cipher CID Algorithm

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

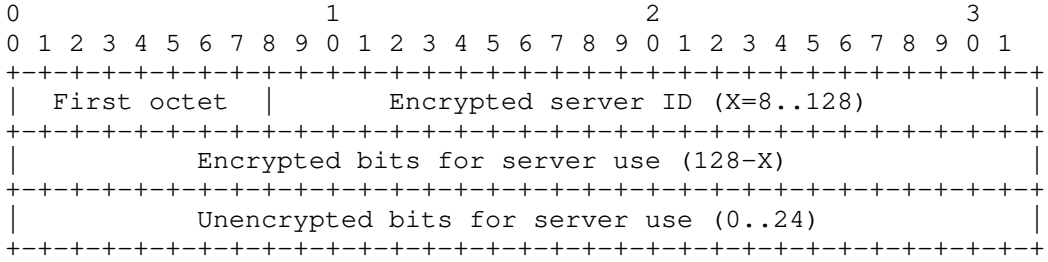

Figure 4: Block Cipher CID Format

4.3.1. Configuration Agent Actions

The configuration agent assigns a server ID to every server in its pool, and determines a server ID length (in octets) sufficiently large to encode all server IDs, including potential future servers. The server ID will start in the second octet of the decrypted connection ID and occupy continuous octets beyond that.

They server ID length MUST be no more than 16 octets and SHOULD be no more than 12 octets, to provide servers adequate space to encode their own opaque data.

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

4.3.2. Load Balancer Actions

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

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

4.3.3. Server Actions

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

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

5. ICMP Processing

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

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

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

6. Retry Service

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

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

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

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

6.1. Common Requirements

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

A retry service MUST forward all packets for a QUIC version it does not understand. Note that if servers support versions the retry service does not, this may increase load on the servers. However, dropping these packets would introduce chokepoints to block deployment of new QUIC versions. Note that future versions of QUIC might not have Retry packets, require different information in Retry, or use different packet type indicators.

6.2. No-Shared-State Retry Service

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

6.2.1. Configuration Agent Actions

The configuration agent distributes a list of QUIC versions to be served by the Retry Service.

6.2.2. Service Requirements

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

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

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

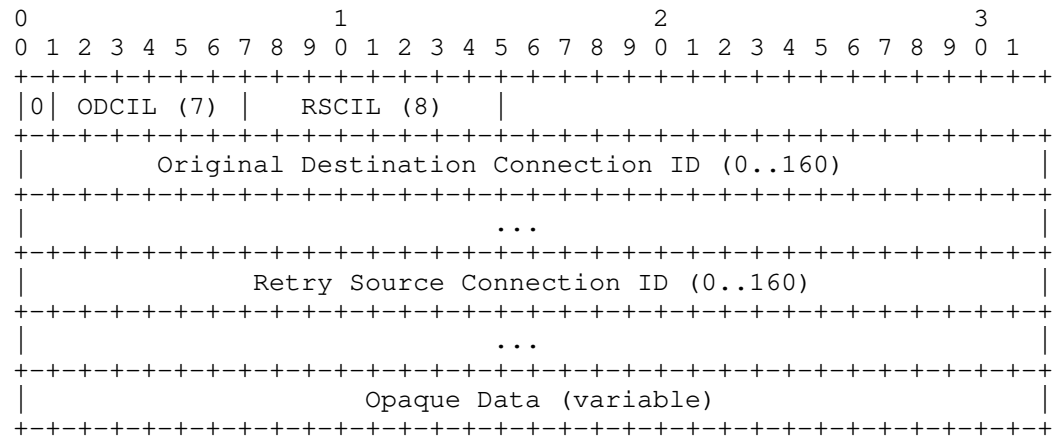

Figure 5: Format of non-shared-state retry service tokens

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

ODCIL: The length of the original destination connection ID from the triggering Initial packet. This is in cleartext to be readable for the server, but authenticated later in the token.

RSCIL: The retry source connection ID length.

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

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

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

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

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

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

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

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

6.2.3. Server Requirements

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

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

If a server receives an Initial Packet with the first bit set to '1', it could be from a server-generated NEW_TOKEN frame and should be processed in accordance with the QUIC specification. If a server receives an Initial Packet with the first bit to '0', it is a Retry token and the server MUST NOT attempt to validate it. Instead, it MUST assume the address is validated and MUST extract the Original Destination Connection ID and Retry Source Connection ID, assuming the format described in Section 6.2.2.

6.3. Shared-State Retry Service

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

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

All tokens, generated by either the server or retry service, MUST use the following format. This format is the cleartext version. On the wire, these fields are encrypted using an AES-ECB cipher and the token key. If the token is not a multiple of 16 octets, the last block is padded with zeroes.

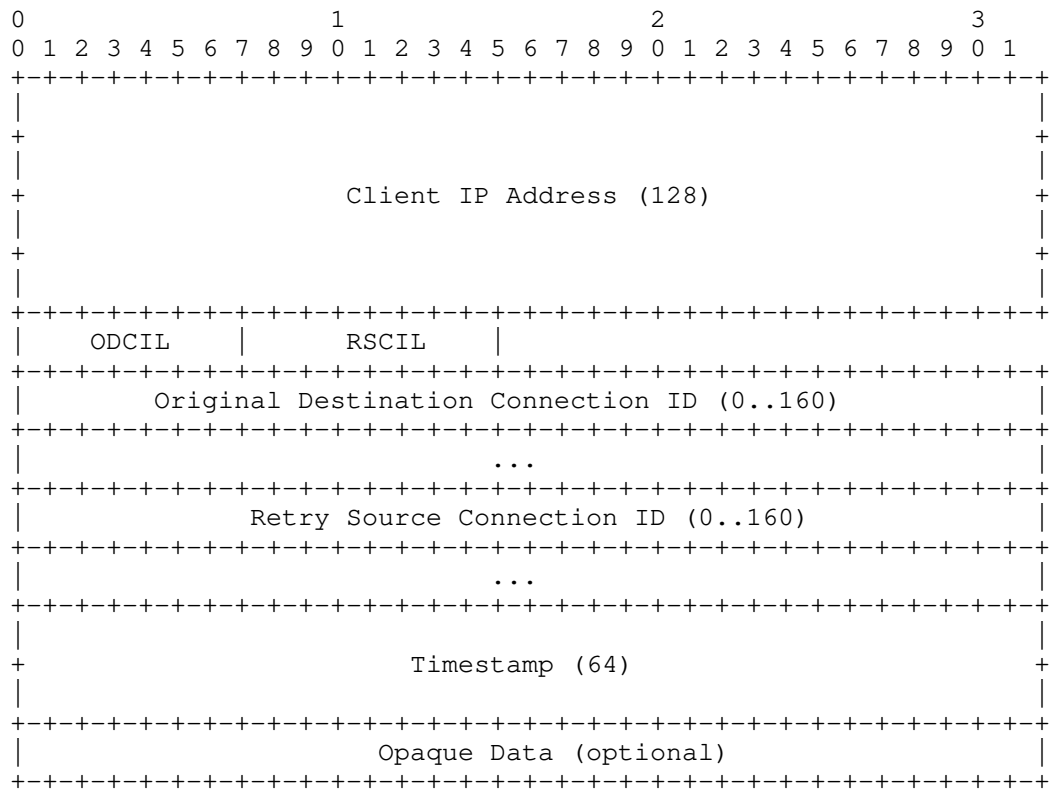

Figure 6: Cleartext format of shared-state retry tokens

The tokens have the following fields:

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

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

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

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

Client IP Address: The source IP address from the triggering Initial packet. The client IP address is 16 octets. If an IPv4 address, the last 12 octets are zeroes. If there is a Network Address Translator (NAT) in the server infrastructure that changes the client IP, the Retry Service MUST either be positioned behind the NAT, or the NAT must have the token key to rewrite the Retry token accordingly.

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

Opaque Data: The server may use this field to encode additional information, such as congestion window, RTT, or MTU. Opaque data MAY also allow servers to distinguish between retry tokens (which trigger use of certain transport parameters) and NEW_TOKEN frame tokens.

6.3.1. Configuration Agent Actions

The configuration agent generates and distributes a "token key" and a list of QUIC versions the service supports. It must also inform the service if a NAT lies between the service and the servers.

6.3.2. Service Requirements

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

In active mode, the service SHOULD decrypt the first 16 octets of incoming tokens. The service SHOULD drop packets with an IP address that does not match, and SHOULD forward packets that do, regardless of the other fields.

However, the service MUST NOT decrypt or validate tokens if there is a NAT between it and the servers.

In inactive mode, the service SHOULD forward all packets to the server so that the server can issue an up-to-date token to the client.

6.3.3. Server Requirements

When issuing Retry or NEW_TOKEN tokens, the server MUST encode the client IP address in the first 16 octets and encrypt that block with the token key. It MAY use any format or encryption for the remainder of the token. However, it MUST include a means of distinguishing service-generated Retry tokens, server-generated Retry tokens (if different), and NEW_TOKEN tokens.

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

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

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

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

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

7. Configuration Requirements

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

The load balancer and server MUST agree on a routing algorithm and the relevant parameters for that algorithm. Each server MUST know its server ID for each configuration, and the load balancer MUST have forwarding instructions for each server ID.

For all algorithms, the load balancer and servers MUST have a common understanding of the server ID length.

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

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

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

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

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

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

The following pseudocode describes the data items necessary to store a full QUIC-LB configuration at the server. It is meant to describe the conceptual range and not specify the presentation of such configuration in an internet packet. The comments signify the range of acceptable values where applicable.

```
    uint2    config_rotation_bits;
    boolean  first_octet_encodes_cid_length;
    enum     { none, non_shared_state, shared_state } retry_service;
    select (retry_service) {
        case none: null;
        case non_shared_state: uint32 list_of_quic_versions[];
        case shared_state: {
            uint32   list_of_quic_versions[];
            uint8    token_key[16];
        } shared_state_config;
    } retry_service_config;
    enum     { none, plaintext, stream_cipher, block_cipher }
                     routing_algorithm;
    select (routing_algorithm) {
        case none: null;
        case plaintext: struct {
            uint8 server_id_length; /* 1..19 */
            uint8 server_id[server_id_length];
        } plaintext_config;
        case stream_cipher: struct {
            uint8  nonce_length; /* 8..16 */
            uint8  server_id_length; /* 1..(19 - nonce_length) */
            uint8  server_id[server_id_length];
            uint8  key[16];
        } stream_cipher_config;
        case block_cipher: struct {
            uint8  server_id_length;
            uint8  server_id[server_id_length];
            uint8  key[16];
        } block_cipher_config;
   } routing_algorithm_config;
```

8. Additional Use Cases

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

8.1. 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.

8.2. 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.

9. Version Invariance of QUIC-LB

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

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

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

Section 4 provides guidance about how load balancers should handle non-compliant DCIDs. This guidance, and the implementation of an algorithm to handle these DCIDs, rests on some assumptions:

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

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

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

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

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

10. Security Considerations

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

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

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

10.1. Attackers not between the load balancer and server

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

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

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

10.2. 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.

10.3. Multiple Configuration IDs

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

10.4. Limited configuration scope

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

To avoid this, the configuration agent SHOULD issue QUIC-LB configurations to mutually distrustful servers that have different keys for encryption algorithms. The load balancers can distinguish these configurations by external IP address, or by assigning different values to the config rotation bits (Section 3.1). Note that either solution has a privacy impact; see Section 10.3.

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

10.5. Stateless Reset Oracle

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

11. IANA Considerations

There are no IANA requirements.

12. References

12.1. Normative References

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

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

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

12.2. 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/info/rfc2119>.

Appendix A. Load Balancer Test Vectors

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

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

This section uses the following abbreviations:

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

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

A.1. Plaintext Connection ID Algorithm

LB configuration: cr_bits 0x0 length_self_encoding: y sid_len 1

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

LB configuration: cr_bits 0x0 length_self_encoding: n sid_len 2

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

LB configuration: cr_bits 0x0 length_self_encoding: y sid_len 3

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

LB configuration: cr_bits 0x0 length_self_encoding: n sid_len 4

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

LB configuration: cr_bits 0x0 length_self_encoding: y sid_len 5

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

A.2. Stream Cipher Connection ID Algorithm

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

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

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

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

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

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

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

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

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

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

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

A.3. Block Cipher Connection ID Algorithm

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

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

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

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

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

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

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

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

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

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

Appendix B. Acknowledgments

Appendix C. Change Log

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

C.1. 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

C.2. since-draft-ietf-quic-load-balancers-03

* Improved Config Rotation text
* Added stream cipher test vectors
* Deleted the Obfuscated CID algorithm

C.3. 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

C.4. 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

C.5. since-draft-ietf-quic-load-balancers-00

* Removed in-band protocol from the document

C.6. Since draft-duke-quic-load-balancers-06

* Switch to IETF WG draft.

C.7. 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

C.8. Since draft-duke-quic-load-balancers-04

* Added standard for retry services

C.9. 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

C.10. 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

C.11. Since draft-duke-quic-load-balancers-01

* Complete rewrite
* Supports multiple security levels
* Lightweight messages

C.12. Since draft-duke-quic-load-balancers-00

* Converted to markdown
* Added variable length connection IDs

Authors' Addresses

Martin Duke
F5 Networks, Inc.

Email: martin.h.duke@gmail.com

Nick Banks
Microsoft

Email: nibanks@microsoft.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: 6 May 2021

M. Kuehlewind
Ericsson
B. Trammell
Google

2 November 2020

Manageability of the QUIC Transport Protocol
draft-ietf-quic-manageability-08

Abstract

This document discusses manageability of the QUIC transport protocol, focusing on caveats impacting network operations involving QUIC traffic. Its intended audience is network operators, as well as content providers that rely on the use of QUIC-aware middleboxes, e.g. for load balancing.

Table of Contents

1. Introduction

QUIC [QUIC-TRANSPORT] is a new transport protocol currently under development in the IETF QUIC working group, focusing on support of semantics as needed for HTTP/2 [QUIC-HTTP]. Based on current deployment practices, QUIC is encapsulated in UDP and encrypted by default. The current version of QUIC integrates TLS [QUIC-TLS] to encrypt all payload data and most control information.

Given that QUIC is an end-to-end transport protocol, all information in the protocol header, even that which can be inspected, is not meant to be mutable by the network, and is therefore integrity-protected. While less information is visible to the network than for TCP, integrity protection can also simplify troubleshooting because none of the nodes on the network path can modify the transport layer information.

This document provides guidance for network operation on the management of QUIC traffic. This includes guidance on how to interpret and utilize information that is exposed by QUIC to the network as well as explaining requirement and assumptions that the QUIC protocol design takes toward the expected network treatment. It also discusses how common network management practices will be impacted by QUIC.

Since QUIC's wire image [WIRE-IMAGE] is integrity protected and not modifiable on path, in-network operations are not possible without terminating the QUIC connection, for instance using a back-to-back proxy. Proxy operations are not in scope for this document. QUIC proxies must be fully-fledged QUIC endpoints, implementing the transport as defined in [QUIC-TRANSPORT] and [QUIC-TLS] as well as proxy-relevant semantics for the application(s) running over QUIC (e.g. HTTP/3 as defined in [QUIC-HTTP]).

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

QUIC is at the moment very much a moving target. This document refers the state of the QUIC working group drafts as well as to changes under discussion, via issues and pull requests in GitHub current as of the time of writing.

1.1. Notational Conventions

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

2. Features of the QUIC Wire Image

In this section, we discusses those aspects of the QUIC transport protocol that have an impact on the design and operation of devices that forward QUIC packets. Here, we are concerned primarily with the unencrypted part of QUIC's wire image [WIRE-IMAGE], which we define as the information available in the packet header in each QUIC packet, and the dynamics of that information. Since QUIC is a versioned protocol, the wire image of the header format can also change from version to version. However, at least the mechanism by which a receiver can determine which version is used and the meaning and location of fields used in the version negotiation process is invariant [QUIC-INVARIANTS].

This document describes only version 1 the QUIC protocol, whose wire image is fully defined in [QUIC-TRANSPORT] and [QUIC-TLS]. Note that features of the wire image described herein and in those documents may change in future versions of the protocol, and cannot be used to identify QUIC as a protocol or to infer the behavior of future versions of QUIC. Section 9.1 provides non-normative guidance on the identification of QUIC version 1 packets compared to other deployed versions at the date if publication.

2.1. QUIC Packet Header Structure

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

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

Short headers are used after connection establishment, and contain only an optional destination connection ID and the spin bit for RTT measurement.

The following information is exposed in QUIC packet headers:

* "fixed bit": the second most significant bit of the first octet most QUIC packets of the current version is currently set to 1, for demultiplexing with other UDP-encapsulated protocols.

* latency spin bit: the third most significant bit of first octet in the short packet header. The spin bit is set by endpoints such that tracking edge transitions can be used to passively observe end-to-end RTT. See Section 3.7.2 for further details.

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

* version number: the version number present in the long header, and identifies the version used for that packet. Note that during Version Negotiation (see Section 2.8, and Section 17.2.1 of [QUIC-TRANSPORT], the version number field has a special value (0x00000000) that identifies the packet as a Version Negotiation packet. QUIC versions that start with 0xff are IETF drafts. QUIC versions that start with 0x0000 are reserved for IETF consensus documents, for example the QUIC version 1 is expected to use version 0x00000001.

* source and destination connection ID: short and long packet headers carry a destination connection ID, a variable-length field that can be used to identify the connection associated with a QUIC packet, for load-balancing and NAT rebinding purposes; see Section 4.3 and Section 2.6. Long packet headers additionally carry a source connection ID. The source connection ID corresponds to the destination connection ID the source would like to have on packets sent to it, and is only present on long packet headers. On long header packets, the length of the connection IDs is also present; on short header packets, the length of the destination connection ID is implicit.

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

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

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

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

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

2.2. Coalesced Packets

Multiple QUIC packets may be coalesced into a UDP datagram, with a datagram carrying one or more long header packets followed by zero or one short header packets. When packets are coalesced, the Length fields in the long headers are used to separate QUIC packets. The length header field is variable length and its position in the header is also variable depending on the length of the source and destination connection ID. See Section 4.6 of [QUIC-TRANSPORT].

2.3. Use of Port Numbers

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

Further, as QUIC has a connection ID, it is also possible to maintain multiple QUIC connections over one 5-tuple. However, if the connection ID is not present in the packet header, all packets of the 5-tuple belong to the same QUIC connection.

2.4. The QUIC handshake

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

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

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

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

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

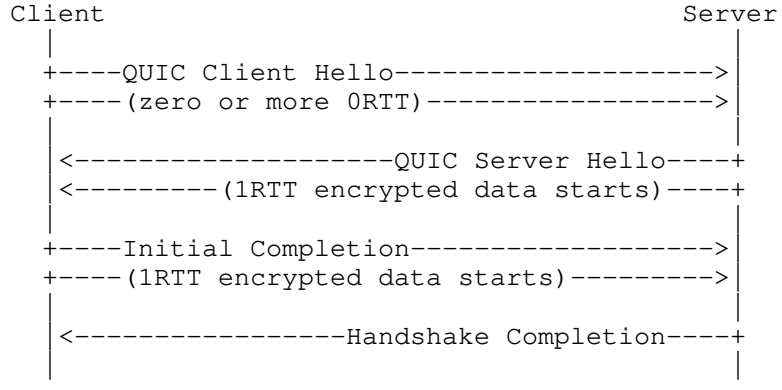

Figure 1: General communication pattern visible in the QUIC handshake

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

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

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

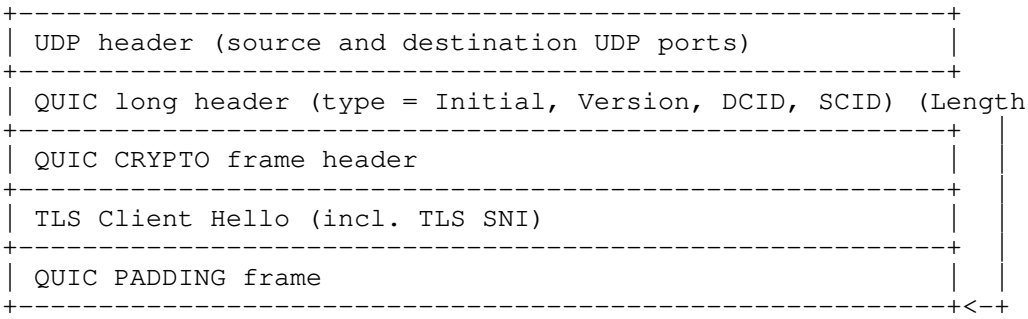

Figure 2: Typical 1-RTT QUIC Client Hello datagram pattern

The Client Hello datagram exposes version number, source and destination connection IDs in the clear. Information in the TLS Client Hello frame, including any TLS Server Name Indication (SNI) present, is obfuscated using the Initial secret. The QUIC PADDING frame shown here may be present to ensure the Client Hello datagram has a minimum size of 1200 octets, to mitigate the possibility of handshake amplification. Note that the location of PADDING is implementation-dependent, and PADDING frames may not appear in the Initial packet in a coalesced packet.

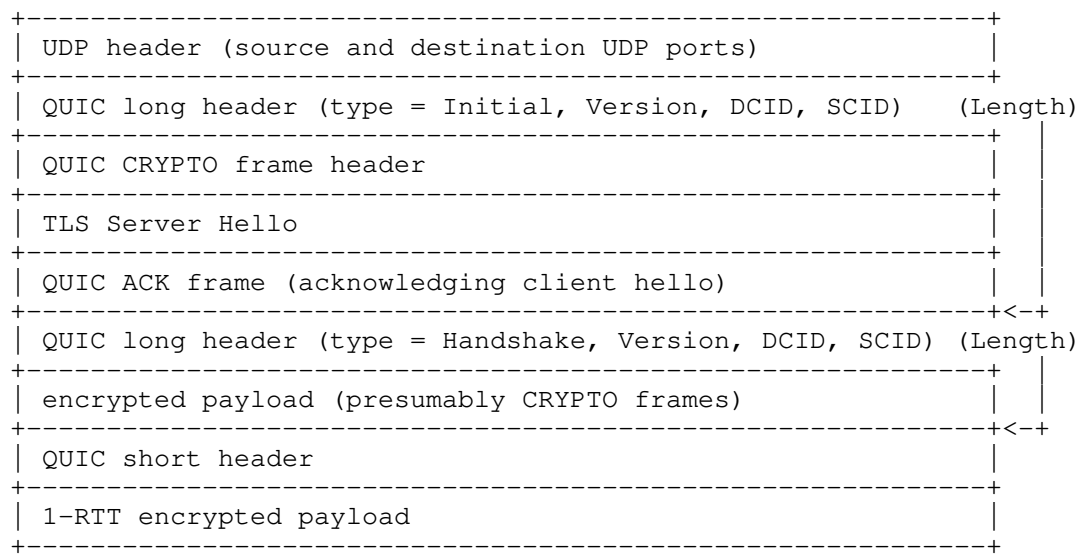

Figure 3: Typical QUIC Server Hello datagram pattern

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

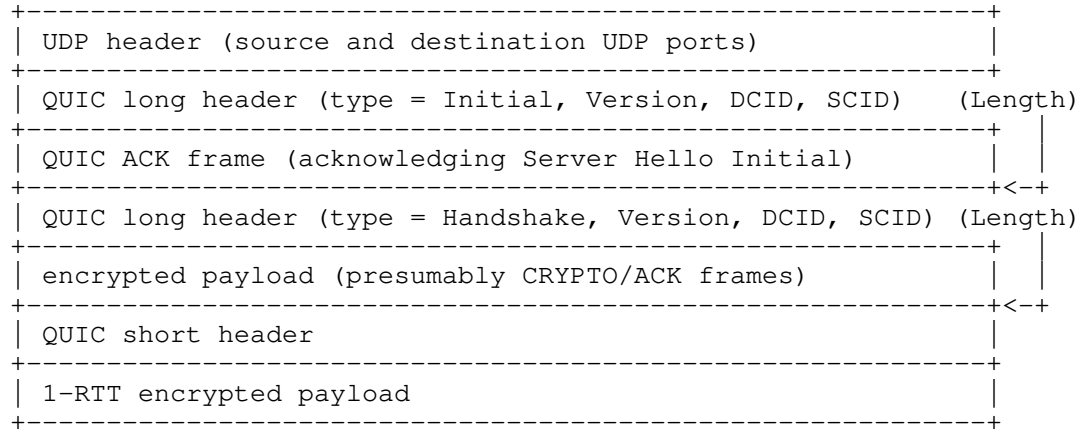

Figure 4: Typical QUIC Initial Completion datagram pattern

The Initial Completion datagram does not expose any additional information; however, recognizing it can be used to determine that a handshake has completed (see Section 3.2), and for three-way handshake RTT estimation as in Section 3.7.

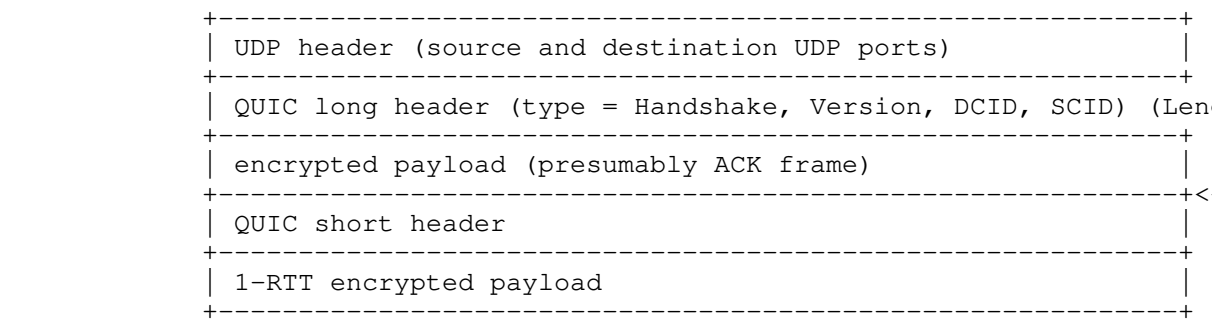

Figure 5: Typical QUIC Handshake Completion datagram pattern

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

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

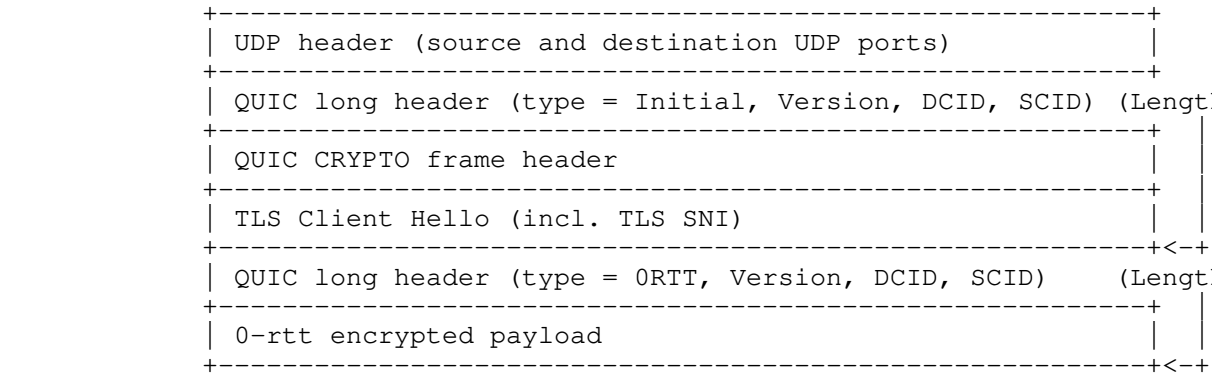

Figure 6: Typical 0-RTT QUIC Client Hello datagram pattern

In a 0-RTT QUIC Client Hello datagram, the PADDING frame is only present if necessary to increase the size of the datagram with 0RTT data to at least 1200 bytes. Additional datagrams containing only 0-RTT protected long header packets may be sent from the client to the server after the Client Hello datagram, containing the rest of the 0-RTT data. The amount of 0-RTT protected data is limited by the initial congestion window, typically around 10 packets [RFC6928].

2.5. Integrity Protection of the Wire Image

As soon as the cryptographic context is established, all information in the QUIC header, including information exposed in the packet header, is integrity protected. Further, information that was sent and exposed in handshake packets sent before the cryptographic context was established are validated later during the cryptographic handshake. Therefore, devices on path MUST NOT change any information or bits in QUIC packet headers, since alteration of header information will lead to a failed integrity check at the receiver, and can even lead to connection termination.

2.6. Connection ID and Rebinding

The connection ID in the QUIC packet headers allows routing of QUIC packets at load balancers on other than five-tuple information, ensuring that related flows are appropriately balanced together; and to allow rebinding of a connection after one of the endpoint's addresses changes - usually the client's, in the case of the HTTP binding. Client and server negotiate connection IDs during the handshake; typically, however, only the server will request a connection ID for the lifetime of the connection. Connection IDs for either endpoint may change during the lifetime of a connection, with the new connection ID being negotiated via encrypted frames. See Section 5.1 of [QUIC-TRANSPORT].

Server-generated connection IDs should seek to obscure any encoding, of routing identities or any other information. Exposing the server mapping would allow linkage of multiple IP addresses to the same host if the server also supports migration. Furthermore, this opens an attack vector on specific servers or pools.

The best way to obscure an encoding is to appear random to observers, which is most rigorously achieved with encryption. Even when encrypted, a scheme could embed the unencrypted length of the Connection ID in the Connection ID itself, instead of remembering it, e.g. by using the first few bits to indicate a certain size of a well-known set of possible sizes with multiple values that indicate the same size but are selected randomly.

[QUIC_LB] further specified possible algorithms to generate Connection IDs at load balancers.

2.7. Packet Numbers

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

2.8. Version Negotiation and Greasing

Version negotiation is not protected, given the used protection mechanism can change with the version. However, the choices provided in the list of version in the Version Negotiation packet will be validated as soon as the cryptographic context has been established. Therefore any manipulation of this list will be detected and will cause the endpoints to terminate the connection.

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

QUIC is expected to evolve rapidly, so new versions, both experimental and IETF standard versions, will be deployed in the Internet more often than with traditional Internet- and transport-layer protocols. Using a particular version number to recognize valid QUIC traffic is likely to persistently miss a fraction of QUIC flows and completely fail in the multi-year timeframe so therefore not recommended.

3. Network-visible information about QUIC flows

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

3.1. Identifying QUIC traffic

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

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

While the second most significant bit (0x40) of the first octet is set to 1 in most QUIC packets of the current version (see Section 2.1), this method of recognizing QUIC traffic is NOT RECOMMENDED. First, it only provides one bit of information and is quite prone to collide with UDP-based protocols other than those that this static bit is meant to allow multiplexing with. Second, this feature of the wire image is not invariant [QUIC-INVARIANTS] and may change in future versions of the protocol, or even be negotiated after handshake via future transport parameters.

3.1.1. Identifying Negotiated Version

An in-network observer assuming that a set of packets belongs to a QUIC flow can infer the version number in use by observing the handshake: an Initial packet with a given version from a client to which a server responds with an Initial packet with the same version implies acceptance of that version.

Negotiated version cannot be identified for flows for which a handshake is not observed, such as in the case of connection migration; however, these flows can be associated with flows for which a version has been identified; see Section 3.4.

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

3.1.2. Rejection of Garbage Traffic

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

3.2. Connection confirmation

Connection establishment uses Initial, Handshake, and Retry packets containing a TLS handshake. Connection establishment can therefore be detected using heuristics similar to those used to detect TLS over TCP. A client using 0-RTT connection may also send data packets in 0-RTT Protected packets directly after the Initial packet containing the TLS Client Hello. Since these packets may be reordered in the network, note that 0-RTT Protected data packets may be seen before the Initial packet.

Note that clients send Initial packets before servers do, servers send Handshake packets before clients do, and only clients send Initial packets with tokens, so the sides of a connection can be generally be confirmed by an on-path observer. An attempted connection after Retry can be detected by correlating the token on the Retry with the token on the subsequent Initial packet.

3.3. Application Identification

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

Work is currently underway in the TLS working group to encrypt the SNI in TLS 1.3 [TLS-ESNI]. If used with QUIC, this would make SNI-based application identification impossible through passive measurement.

3.3.1. Extracting Server Name Indication (SNI) Information

If the SNI is not encrypted it can be derived from the QUIC Initial packet by calculating the Initial Secret to decrypt the packet payload and parse the QUIC CRYPTO Frame containing the TLS ClientHello.

As both the initial salt for the Initial Secret as well as CRYPTO frame itself are version-specific, the first step is always to parse the version number (second to sixth byte of the long header). Note that only long header packets carry the version number, so it is necessary to also check the if first bit of the QUIC packet is set to 1, indicating a long header.

Note, that proprietary QUIC versions, that have been deployed before standardization, might not set the first bit in a QUIC long header packets to 1. To parse these versions example code is provided in the appendix (see Section 9.1), however, it is expected that these versions will gradually disappear over time.

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

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

Finally after decryption, the Initial Client packet can be parsed to detect the CRYPTO frame that contains the TLS Client Hello, which then can be respectively parsed similar as for all other TLS connections. The Initial client packet may contain other frames, so the first byte of each frame need to be checked to identify the frame type and the skip over the frame. Note that the length of the frames is dependent on the frame type. Usually for QUIC version 1, the packet is expected to only carry the CRYPTO frame and optionally padding frames. However, padding which is one byte of zeros, may also occur before or after the CRYPTO frame.

3.4. Flow association

The QUIC Connection ID (see Section 2.6) is designed to allow an on-path device such as a load-balancer to associate two flows as identified by five-tuple when the address and port of one of the endpoints changes; e.g. due to NAT rebinding or server IP address migration. An observer keeping flow state can associate a connection ID with a given flow, and can associate a known flow with a new flow when when observing a packet sharing a connection ID and one endpoint address (IP address and port) with the known flow.

However, since the connection ID may change multiple times during the lifetime of a flow, and the negotiation of connection ID changes is encrypted, packets with the same 5-tuple but different connection IDs may or may not belong to the same connection.

The connection ID value should be treated as opaque; see Section 4.3 for caveats regarding connection ID selection at servers.

3.5. Flow teardown

QUIC does not expose the end of a connection; the only indication to on-path devices that a flow has ended is that packets are no longer observed. Stateful devices on path such as NATs and firewalls must therefore use idle timeouts to determine when to drop state for QUIC flows, see further section Section 4.1.

3.6. Flow symmetry measurement

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

3.7. Round-Trip Time (RTT) Measurement

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

3.7.1. Measuring initial RTT

In the common case, the delay between the Initial packet containing the TLS Client Hello and the Handshake packet containing the TLS Server Hello represents the RTT component on the path between the observer and the server. The delay between the TLS Server Hello and the Handshake packet containing the TLS Finished message sent by the client represents the RTT component on the path between the observer and the client. While the client may send 0-RTT Protected packets after the Initial packet during 0-RTT connection re-establishment, these can be ignored for RTT measurement purposes.

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

3.7.2. Using the Spin Bit for Passive RTT Measurement

The spin bit provides an additional method to measure per-flow RTT from observation points on the network path throughout the duration of a connection. Endpoint participation in spin bit signaling is optional in QUIC. That is, while its location is fixed in this version of QUIC, an endpoint can unilaterally choose to not support "spinning" the bit. Use of the spin bit for RTT measurement by devices on path is only possible when both endpoints enable it. Some endpoints may disable use of the spin bit by default, others only in specific deployment scenarios, e.g. for servers and clients where the RTT would reveal the presence of a VPN or proxy. To avoid making these connections identifiable based on the usage of the spin bit, it is recommended that all endpoints randomly disable "spinning" for at least one eighth of connections, even if otherwise enabled by default. An endpoint not participating in spin bit signaling for a given connection can use a fixed spin value for the duration of the connection, or can set the bit randomly on each packet sent.

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

Note that this measurement, as with passive RTT measurement for TCP, includes any transport protocol delay (e.g., delayed sending of acknowledgements) and/or application layer delay (e.g., waiting for a response to be generated). It therefore provides devices on path a good instantaneous estimate of the RTT as experienced by the application. A simple linear smoothing or moving minimum filter can be applied to the stream of RTT information to get a more stable estimate.

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

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

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

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

4. Specific Network Management Tasks

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

4.1. Stateful treatment of QUIC traffic

Stateful treatment of QUIC traffic (e.g., at a firewall or NAT middlebox) is possible through QUIC traffic and version identification (Section 3.1) and observation of the handshake for connection confirmation (Section 3.2). The lack of any visible end-of-flow signal (Section 3.5) means that this state must be purged either through timers or through least-recently-used eviction, depending on application requirements.

[RFC4787] recommends a 2 minute timeout interval for UDP, however, often timer are lower in the range of 15 to 30 second. In constrast [RFC5382] recommends a timeout of more than 2 hours for TCP, given TCP is a connection-oriented protocol with well defined closure semantics. For network devices that are QUIC-aware, it is recommended to also use longer timeouts for QUIC traffic, as QUIC is connection-oriented and as such a handshake packet from the server indicates the willingness of the server to communicate with the client.

The QUIC header optionally contains a Connection ID which can be used as additional entropy beyond the 5-tuple, if needed. The QUIC handshake needs to be observed in order to understand whether the Connection ID is present and what length it has. However, Connection IDs may be renegotiated during a connection, and this renegotiation is not visible to the path. Keying state off the Connection ID may therefore cause undetectable and unrecoverable loss of state in the middle of a connection. Use of Connection ID specifically discouraged for NAT applications.

4.2. Passive network performance measurement and troubleshooting

Limited RTT measurement is possible by passive observation of QUIC traffic; see Section 3.7. No passive measurement of loss is possible with the present wire image. Extremely limited observation of upstream congestion may be possible via the observation of CE markings on ECN-enabled QUIC traffic.

4.3. Server cooperation with load balancers

In the case of content distribution networking architectures including load balancers, the connection ID provides a way for the server to signal information about the desired treatment of a flow to the load balancers. Guidance on assigning connection IDs is given in [QUIC-APPLICABILITY].

4.4. DDoS Detection and Mitigation

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

Key to successful DDoS mitigation is efficient classification of this traffic in the mitigation environment. Limited first-packet garbage detection as in Section 3.1.2 and stateful tracking of QUIC traffic as in Section 4.1 above may be useful during classification.

Note that the use of a connection ID to support connection migration renders 5-tuple based filtering insufficient and requires more state to be maintained by DDoS defense systems. For the common case of NAT rebinding, DDoS defense systems can detect a change in client's endpoint address by linking flows based on the first 8 bytes of the server's connection IDs, provided the server is using at least 8-bytes-long connection IDs. QUIC's linkability resistance ensures that a deliberate connection migration is accompanied by a change in the connection ID and necessitate that connection ID aware DDoS defense system must have the same information about connection IDs as the load balancer [I-D.ietf-quic-load-balancers]. This may be complicated where mitigation and load balancing environments are logically separate.

It is questionable whether connection migrations must be supported during a DDoS attack. If the connection migration is not visible to the network that performs the DDoS detection, an active, migrated QUIC connection may be blocked by such a system under attack. As soon as the connection blocking is detected by the client, the client may rely on the fast resumption mechanism provided by QUIC. When clients migrate to a new path, they should be prepared for the migration to fail and attempt to reconnect quickly.

4.5. UDP Policing

Today, UDP is the most prevalent DDoS vector, since it is easy for compromised non-admin applications to send a flood of large UDP packets (while with TCP the attacker gets throttled by the congestion controller) or to craft reflection and amplification attacks. Networks should therefore be prepared for UDP flood attacks on ports used for QUIC traffic. One possible response to this threat is to police UDP traffic on the network, allocating a fixed portion of the network capacity to UDP and blocking UDP datagram over that cap.

The recommended way to police QUIC packets is to either drop them all or to throttle them based on the hash of the UDP datagram's source and destination addresses, blocking a portion of the hash space that corresponds to the fraction of UDP traffic one wishes to drop. When the handshake is blocked, QUIC-capable applications may failover to TCP (at least applications using well-known UDP ports). However, blindly blocking a significant fraction of QUIC packets will allow many QUIC handshakes to complete, preventing a TCP failover, but the connections will suffer from severe packet loss.

4.6. Distinguishing acknowledgment traffic

Some deployed in-network functions distinguish pure-acknowledgment (ACK) packets from packets carrying upper-layer data in order to attempt to enhance performance, for example by queueing ACKs differently or manipulating ACK signaling. Distinguishing ACK packets is trivial in TCP, but not supported by QUIC, since acknowledgment signaling is carried inside QUIC's encrypted payload, and ACK manipulation is impossible. Specifically, heuristics attempting to distinguish ACK-only packets from payload-carrying packets based on packet size are likely to fail, and are emphatically NOT RECOMMENDED.

4.7. QoS support and ECMP

[EDITOR'S NOTE: this is a bit speculative; keep?]

QUIC does not provide any additional information on requirements on Quality of Service (QoS) provided from the network. QUIC assumes that all packets with the same 5-tuple {dest addr, source addr, protocol, dest port, source port} will receive similar network treatment. That means all stream that are multiplexed over the same QUIC connection require the same network treatment and are handled by the same congestion controller. If differential network treatment is desired, multiple QUIC connections to the same server might be used, given that establishing a new connection using 0-RTT support is cheap and fast.

QoS mechanisms in the network MAY also use the connection ID for service differentiation, as a change of connection ID is bound to a change of address which anyway is likely to lead to a re-route on a different path with different network characteristics.

Given that QUIC is more tolerant of packet re-ordering than TCP (see Section 2.7), Equal-cost multi-path routing (ECMP) does not necessarily need to be flow based. However, 5-tuple (plus eventually connection ID if present) matching is still beneficial for QoS given all packets are handled by the same congestion controller.

5. IANA Considerations

This document has no actions for IANA.

6. Security Considerations

Supporting manageability of QUIC traffic inherently involves tradeoffs with the confidentiality of QUIC's control information; this entire document is therefore security-relevant.

7. Contributors

Dan Druta contributed text to Section 4.4. Igor Lubashev contributed text to Section 4.3 on the use of the connection ID for load balancing. Marcus Ilhar contributed text to Section 3.7 on the use of the spin bit. The pseudo provided in the appendix is based on input provided by David Schinazi.

8. Acknowledgments

Thanks to Martin Thomson and Martin Duke for contributing by reviewing and providing text proposals.

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

9. Appendix

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

9.1. Distinguishing IETF QUIC and Google QUIC Versions

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

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

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

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

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

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

9.2. Extracting the CRYPTO frame

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

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

10. References

10.1. 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/info/rfc2119>.

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

10.2. Informative References

[Ding2015] Ding, H. and M. Rabinovich, "TCP Stretch Acknowledgments and Timestamps - Findings and Impliciations for Passive RTT Measurement (ACM Computer Communication Review)", July 2015, <http://www.sigcomm.org/sites/default/files/ccr/papers/2015/July/0000000-0000002.pdf>.

[DOTS-ARCH] Mortensen, A., Reddy.K, T., Andreasen, F., Teague, N., and R. Compton, "Distributed-Denial-of-Service Open Threat

Signaling (DOTS) Architecture", Work in Progress, Internet-Draft, draft-ietf-dots-architecture-18, 6 March 2020, <http://www.ietf.org/internet-drafts/draft-ietf-dots-architecture-18.txt>.

[I-D.ietf-quic-load-balancers]
Duke, M. and N. Banks, "QUIC-LB: Generating Routable QUIC Connection IDs", Work in Progress, Internet-Draft, draft-ietf-quic-load-balancers-05, 30 October 2020, <http://www.ietf.org/internet-drafts/draft-ietf-quic-load-balancers-05.txt>.

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

[QUIC-APPLICABILITY]
Kuehlewind, M. and B. Trammell, "Applicability of the QUIC Transport Protocol", Work in Progress, Internet-Draft, draft-ietf-quic-applicability-07, 8 July 2020, <http://www.ietf.org/internet-drafts/draft-ietf-quic-applicability-07.txt>.

[QUIC-HTTP]
Bishop, M., "Hypertext Transfer Protocol Version 3 (HTTP/3)", Work in Progress, Internet-Draft, draft-ietf-quic-http-32, 20 October 2020, <http://www.ietf.org/internet-drafts/draft-ietf-quic-http-32.txt>.

[QUIC-INVARIANTS]
Thomson, M., "Version-Independent Properties of QUIC", Work in Progress, Internet-Draft, draft-ietf-quic-invariants-11, 24 September 2020, <http://www.ietf.org/internet-drafts/draft-ietf-quic-invariants-11.txt>.

[QUIC-TLS] Thomson, M. and S. Turner, "Using TLS to Secure QUIC", Work in Progress, Internet-Draft, draft-ietf-quic-tls-32, 20 October 2020, <http://www.ietf.org/internet-drafts/draft-ietf-quic-tls-32.txt>.

[QUIC-TRANSPORT]
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", Work in Progress, Internet-Draft, draft-ietf-quic-transport-32, 20 October 2020, <http://www.ietf.org/internet-drafts/draft-ietf-quic-transport-32.txt>.

[QUIC_LB] Duke, M. and N. Banks, "QUIC-LB: Generating Routable QUIC Connection IDs", Work in Progress, Internet-Draft, draft-ietf-quic-load-balancers-05, 30 October 2020, <http://www.ietf.org/internet-drafts/draft-ietf-quic-load-balancers-05.txt>.

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

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

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

[RFC6928] Chu, J., Dukkipati, 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>.

[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>.

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

[TLS-ESNI] Rescorla, E., Oku, K., Sullivan, N., and C. Wood, "TLS Encrypted Client Hello", Work in Progress, Internet-Draft, draft-ietf-tls-esni-08, 16 October 2020, <http://www.ietf.org/internet-drafts/draft-ietf-tls-esni-08.txt>.

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

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

Authors' Addresses

Mirja Kuehlewind
Ericsson

Email: mirja.kuehlewind@ericsson.com

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

Email: ietf@trammell.ch

Internet Engineering Task Force
Internet-Draft
Intended status: Informational
Expires: November 19, 2020

N. Kuhn
CNES
E. Stephan
Orange
G. Fairhurst
T. Jones
University of Aberdeen

May 18, 2020

Transport parameters for 0-RTT connections
draft-kuhn-quic-0rtt-bdp-07

Abstract

0-RTT mechanisms reduce the time it takes for the first bytes of application data to be processed in a transport connection and can greatly reduce connection latency during setup. The 0-RTT transport features described by quic-transport help clients establish secure connections with a minimal number of round-trips.

This document describes a generic method to exchange path parameters relating to transport. The additional transport parameters can help a connection that continues after an interruption or restarts by sharing connection properties. They can be used to increase the performance for a path with large RTT.

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 November 19, 2020.

Table of Contents

1. Introduction

Each transport connection typically starts without knowledge of the path between the endpoints. Transport protocols use implicit signals from the network to discover the properties of the path. This information is used to adapt the transport mechanisms to the network path. For example, an Internet transport endpoint is unable to determine a safe rate at which to start or continue their transmission, and uses slow-start to determine a safe rate. This applies to the 1-RTT mode of QUIC.

QUIC supports the sending of data in two different modes, after the transport handshake has completed, 1-RTT mode, and sending data along with handshake packets, 0-RTT mode. Using 0-RTT data an application is able to send transport parameters with the handshake packets, making it possible to reduce the latency of the connection setup.

In 0-RTT mode, a QUIC server must store a copy of a number of flow control related transport parameters, or receives an integrity-

protected copy of these values in the ticket the client includes in the first message of the handshake, to enable the use of 0-RTT data. The setting or omission of one of these parameters can result in QUIC creating a connection, but flow control can still prevent any data being sent by the client.

For 0-RTT data to be sent, the QUIC server must record the values of:

- initial_max_data
- initial_max_stream_data_bidi_local
- initial_max_stream_data_bidi_remote
- initial_max_stream_data_uni
- initial_max_streams_bidi
- initial_max_streams_uni

These values set the flow control limits within which a connection must operate. The server has to store these parameters for a client to send data when resuming during 0-RTT. The stored values are used for any data that is transmitted before the handshake has completed and 1-RTT data is able to be sent on the connection. Once the handshake has completed, these values are discarded and the values established during the handshake are used.

This document proposes an extension to the transport parameters that are shared during the 0-RTT phase to allow resumption using additionnal transport and connection properties that were discovered in previous connections. These additional parameters aim to provide faster startup of flows and better buffer management. The BDP metadata extension is proposed and candidate parameters are discussed in Section 3.

2. Motivation

Reducing the number of round-trips required to start a connection is an important way to reduce setup time and lower overall connection latency. 0-RTT mechanisms that allow a client to feed requests to a server in the first RTT do not alone improve the total time-to-service. The BDP extension described in this document aims to improve traffic delivery by allowing the connection to short-cut slow RTT-based processes that grow connection parameters.

Currently each side has a proprietary solution to measure and to store path characteristics. Recalling the parameters of a previous connection would allow:

- Server and client to re-initialise the state from previously established parameters (i.e., minRTT, MTU, bottleneck capacity, etc.)
- The client to prepare the right resources
- The server to adapt to non-default path characteristics

There is an interest in sharing transport information across multiple connections. As one example, [I-D.ietf-tcpm-2140bis] considers the sharing of transport parameters between connections originating from the same host. The proposal in this document has the advantage of storing the information at the client and not requiring the server to retain additional state for each client.

3. BDP metadata parameters

Section 7.3.1 of [I-D.ietf-quic-transport] describes the parameters that must be remembered if a client wishes to send 0-RTT data. Both endpoints store the value of the server transport parameters from a connection and apply them to any 0-RTT packets that are sent in subsequent connections to the same peer. Of the six mandatory parameters, only initial_max_data improves the time-to-service of the 0-RTT connection. The BDP metadata extension augments the list of server transport parameters that are shared with the client to improve the time-to-service and save resources such as CPU, memory and power.

The BDP extension proposes three new parameters that help a connection startup in a minimal number of RTTs:

- recon_bytes_in_flight (0x000X): The bytes in flight measured on the previous connection by the server. Integer number of bytes. Using the bytes_in_flight defined in [I-D.ietf-quic-recovery], recon_bytes_in_flight can be set to bytes_in_flight.
- recon_min_rtt (0x000X): The minimum RTT measured on the previous connection by the server. Integer number of milliseconds. Using the min_rtt defined in [I-D.ietf-quic-recovery], recon_min_rtt can be set to min_rtt. The min_rtt parameter may not track a decreasing RTT: the min_rtt that is reported here may not be the actual minimum RTT measured during the 1-RTT connection, but still reflects the characteristics of the latency on the network.

- recon_max_pkt_number (0x000X): The maximum amount of packets that the server is allowed to send when reconnecting. Integer unit of packets.

4. Extension activation

The BDP extension is protected by the same mechanism that protects the exchange of the 0-RTT transport parameters. A client that activates 0-RTT data sends back the transport parameters received from the server during the previous connection (see Section 7.3.1 of [I-D.ietf-quic-transport]).

The client reads the parameters in the BDP metadata extension, but can not change them.

Accept: A client MAY use the extension parameters. Then, it activates ingress optimization and sends back the transport parameters of the BDP metadata extension that it received from the server during the previous connection.

Refuse: A client could choose not to use there parameters. Then, it does not support ingress optimisation and drops the extension signal. A client that disagrees with the extension parameters received from the server refuses the optimization.

5. Discussion

The recon_bytes_in_flight parameter is higher than the number of bytes in the actual BDP since it may include bytes in buffers along the path.

The recon_bytes_in_flight parameter is an indication of the end-to-end BDP that is experienced by the congestion and flow control. If the recon_bytes_in_flight is high, the server may decide to increase the maximum amount of packets it will send when reconnecting using the recon_max_pkt_number parameter.

The maximum number of initial data packets that can be sent without acknowledgment needs to be chosen to avoid congestion collapse. An example of a server solution is proposed in Appendix A. In short:

- recon_bytes_in_flight indicates the characteristics of the network underneath to both peers that can adapt their buffer sizes or parameter tuning.

- recon_min_rtt lets both a client and a server know the minimum RTT of the previous connection. Parameters can then be adapted (e.g.,

to adapt usage of kInitialRtt in "Setting the Loss Detection Timer", see section of [I-D.ietf-quic-recovery]).

- recon_max_pkt_number lets the server warn the client that it may increase the amount of packets that it expects to send when reconnecting. This value is negotiated with the client and result in better buffer management and reduced flow start up.

Other parameters can contribute to the optimization of 0-RTT connection. There are good candidates, like max_ack_delay, in the Appendix of [I-D.ietf-quic-recovery].

6. Acknowledgments

The authors would like to thank Gabriel Montenegro, Patrick McManus, Ian Swett, Igor Lubashev, Christian Huitema and Tom Jones for their fruitful comments on earlier versions of this document.

7. IANA Considerations

TBD: Text is required to register the extension BDP_metadata field. Parameters are registered using the procedure defined in [I-D.ietf-quic-transport].

8. Security Considerations

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 the 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 [I-D.ietf-quic-tls]. It provides the server with a way to check the parameters proposed by the client are those that the server sent to the client during the previous connexion.

9. Informative References

[I-D.cardwell-iccrg-bbr-congestion-control]
Cardwell, N., Cheng, Y., Yeganeh, S., and V. Jacobson, "BBR Congestion Control", draft-cardwell-iccrg-bbr-congestion-control-00 (work in progress), July 2017.

[I-D.ietf-quic-recovery]
Iyengar, J. and I. Swett, "QUIC Loss Detection and Congestion Control", draft-ietf-quic-recovery-27 (work in progress), March 2020.

[I-D.ietf-quic-tls]
Thomson, M. and S. Turner, "Using TLS to Secure QUIC", draft-ietf-quic-tls-27 (work in progress), February 2020.

[I-D.ietf-quic-transport]
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-27 (work in progress), February 2020.

[I-D.ietf-tcpm-2140bis]
Touch, J., Welzl, M., and S. Islam, "TCP Control Block Interdependence", draft-ietf-tcpm-2140bis-05 (work in progress), April 2020.

[RFC2914] Floyd, S., "Congestion Control Principles", BCP 41, RFC 2914, DOI 10.17487/RFC2914, September 2000, <https://www.rfc-editor.org/info/rfc2914>.

[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>.

Appendix A. Example of server solution

This section details a solution at the server to safely increase the maximum amount of packets that the server sends when receiving a 0-RTT packet from a client.

The initial window is considered a safe starting point for an unknown path to avoid adding congestion to a congested network. The general assumption is that a path does not currently suffer persistent congestion, and therefore the initial window is applicable until feedback about the path is received. The resulting initial sending rate is only tentative until the capacity is confirmed to be available. If there is loss within this initial transmission, then this could be evidence that the path is congested, and the sender needs to adjust to this congestion.

Significant loss could be an indication of congestion collapse - i.e. persistent loss, requiring back-off of the sending rate.

If however, the reception of IW is confirmed for the first RTT of data, and also the path is determined to be similar to that of a recent previous session (e.g., similar RTT), the method permits the sender to use the previous path information as an input to help determine a new safe rate. One possibility is to immediately jump to a new sending rate that is derived from the previously sustained

rate. This follows the ideas of [I-D.cardwell-iccrg-bbr-congestion-control] and [RFC4782].

The QoS mechanisms that are deployed in the networks can help in prevent the congestion collapse from occurring. However, the sender must provide a significant reduction if there is evidence of potential congestion collapse [RFC2914] from his point of view. Precautions can be taken to guarantee that it is reasonably safe to jump to a high sending rate : measuring that network conditions did not change, allowing some space for other flows that have started and pace transmission of packets. The sender might need to rapidly reduce its rate, if the higher sending rate does not prove to be supported. It if is supported, the sender can resume standard congestion control.

Authors' Addresses

Nicolas Kuhn
CNES

Email: nicolas.kuhn@cnes.fr

Emile Stephan
Orange

Email: emile.stephan@orange.com

Gorry Fairhurst
University of Aberdeen

Email: gorry@erg.abdn.ac.uk

Tom Jones
University of Aberdeen

Email: tom@erg.abdn.ac.uk

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

R. Marx
Hasselt University

2 November 2020

QUIC and HTTP/3 event definitions for qlog
draft-marx-qlog-event-definitions-quic-h3-02

Abstract

This document describes concrete qlog event definitions and their metadata for QUIC and HTTP/3-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 6 May 2021.

Copyright Notice

Table of Contents

1. Introduction

This document describes the values of the qlog name ("category" + "event") and "data" fields and their semantics for the QUIC and HTTP/3 protocols. This document is based on draft-29 of the QUIC and HTTP/3 I-Ds QUIC-TRANSPORT [QUIC-HTTP] and draft-16 of the QPACK I-D [QUIC-QPACK].

Feedback and discussion welcome at https://github.com/quiclog/internet-drafts (https://github.com/quiclog/internet-drafts). 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 examples and data definitions in ths document are expressed in a custom data definition language, inspired by JSON and TypeScript, and 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 and HTTP/3 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 connectivity, transport, http) 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 "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. Importance

Many of the events defined in this document map directly to concepts seen in the QUIC and HTTP/3 documents, while others act as aggregating events that combine data from several possible protocol behaviours or code paths into one. This is done 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. Similarly, we prevent logging duplicate packet data as much as possible. As such, especially packet header value updates are split out into separate events (for example spin_bit_updated, connection_id_updated), as they are expected to change sparingly.

Consequently, many events that can be directly inferred from data on the wire (for example flow control limit changes) if the implementation is bug-free, are currently not explicitly defined as stand-alone events. Exceptions can be made for common events that benefit from being easily identifiable or individually logged (for example the "packets_acked" event). This can in turn give rise to separate events logging similar data, where it is not always clear which event should be logged (for example the separate "connection_started" event, whereas the more general "connection_state_updated" event also allows indicating that a connection was started).

To aid in this decision making, each event has an "importance indicator" with one of three values, in decreasing order of importance and exptected usage:

* Core

* Base

* Extra

The "Core" events are the events that SHOULD be present in all qlog files. These are mostly 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 frame-level 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 "packet_received" events and thus also not which (if any) ACK frames the packet contain, SHOULD log "packets_acked" events instead.

Finally, for event types who's data (partially) overlap with other event types' definitions, where necessary this document includes guidance on which to use in specific situations.

2.2. Custom fields

Note that implementers are free to define new category and event types, 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 SHOULD NOT however 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.

3. 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).

4. QUIC and HTTP/3 fields

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, etc.).

The value of the "protocol_type" qlog field MUST be "QUIC_HTTP3".

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).

4.1. 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. As such, they are grouped in a separate optional field called "raw" of type RawInfo (where applicable).

```
class RawInfo {
    length?:uint64; // full packet/frame length, including header and AEAD authen
tication tag lengths (where applicable)
    payload_length?:uint64; // length of the packet/frame payload, excluding AEAD
 tag. For many control frames, this will have a value of zero

    data?:bytes; // full packet/frame contents, including header and AEAD authent
ication tag (where applicable)
}
```

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

Note: There is intentionally no explicit header_length field in RawInfo. QUIC and HTTP/3 use many Variable-Length Integer Encoded (VLIE) values in their packet and frame headers, which are of a dynamic length. Note too that because of this, we cannot deterministally reconstruct the header encoding/length from qlog data, as implementations might not necessarily employ the most efficient VLIE scheme for all values. As such, it is typically easier to log just the total packet/frame length and the payload length. The header length can be calculated by tools as:

For QUIC packets: header_length = length - payload_length - aead_tag_length

For QUIC and HTTP/3 frames: header_length = length - payload_length

For UDP datagrams: header_length = length - payload_length

Note: In some cases, the length fields are also explicitly reflected inside of frame/packet headers. For example, the QUIC STREAM frame has a "length" field indicating its payload size. Similarly, all HTTP/3 frames include their explicit payload lengths in the frame header. Finally, 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 and HTTP/3 specifications to qlog, making it easier for users to interpret.

Note: as described in [QLOG-MAIN], the RawInfo:data field can be truncated for privacy or security purposes (for example excluding payload data). In this case, the length properties should still indicate the non-truncated lengths.

5. 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".

5.1. connectivity

5.1.1. server_listening

Importance: Extra

Emitted when the server starts accepting connections.

Data:

```
{
    ip_v4?: IPAddress,
    ip_v6?: IPAddress,
    port_v4?: uint32,
    port_v6?: uint32,

    retry_required?:boolean // the server will always answer client initials with
 a retry (no 1-RTT connection setups by choice)
}
```

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

5.1.2. 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.

Data:

```
{
    ip_version?: "v4" | "v6",
    src_ip?: IPAddress,
    dst_ip?: IPAddress,

    protocol?: string, // transport layer protocol (default "QUIC")
    src_port?: uint32,
    dst_port?: uint32,

    src_cid?: bytes,
    dst_cid?: bytes,

}
```

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

5.1.3. 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.

```
{
    owner?:"local"|"remote", // which side closed the connection

    connection_code?:TransportError | CryptoError | uint32,
    application_code?:ApplicationError | uint32,
    internal_code?:uint32,

    reason?:string
}
```

Triggers: * clean * handshake_timeout * idle_timeout * error // this is called the "immediate close" in the QUIC specification * stateless_reset * version_mismatch * application // for example HTTP/3's GOAWAY frame

5.1.4. 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.

Data:

```
{
    owner: "local" | "remote",

    old?:bytes,
    new?:bytes,
}
```

5.1.5. 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.

Data:

```
{
    state: boolean
}
```

5.1.6. connection_retried

TODO

5.1.7. 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.

Data: ~~~ { old?: ConnectionState | SimpleConnectionState, new: ConnectionState | SimpleConnectionState }

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

enum SimpleConnectionState { attempted, handshake_started, handshake_confirmed, closed } ~~~

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 _(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.

5.1.8. 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

5.2. security

5.2.1. 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

Data:

```
{
    key_type:KeyType,
    old?:bytes,
    new:bytes,
    generation?:uint32 // needed for 1RTT key updates
}
```

Triggers:

* "tls" // (e.g., initial, handshake and 0-RTT keys are generated by TLS)

* "remote_update"

* "local_update"

5.2.2. key_retired

Importance: Base

Data:

```
{
    key_type:KeyType,
    key?:bytes,
    generation?:uint32 // needed for 1RTT key updates
}
```

Triggers:

* "tls" // (e.g., initial, handshake and 0-RTT keys are dropped implicitly)

* "remote_update"

* "local_update"

5.3. transport

5.3.1. 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.

Data:

```
{
    server_versions?:Array<bytes>,
    client_versions?:Array<bytes>,
    chosen_version?:bytes
}
```

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

5.3.2. 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.

Data:

```
{
    server_alpns?:Array<string>,
    client_alpns?:Array<string>,
    chosen_alpn?:string
}
```

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.

5.3.3. 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.

Data:

```
{
    owner?:"local" | "remote",

    resumption_allowed?:boolean, // valid session ticket was received
    early_data_enabled?:boolean, // early data extension was enabled on the TLS l
ayer
    tls_cipher?:string, // (e.g., "AES_128_GCM_SHA256")
    aead_tag_length?:uint8, // depends on the TLS cipher, but it's easier to be e
xplicit. Default value is 16

    // transport parameters from the TLS layer:
    original_destination_connection_id?:bytes,
    initial_source_connection_id?:bytes,
    retry_source_connection_id?:bytes,
    stateless_reset_token?:Token,
    disable_active_migration?:boolean,

    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
}

interface PreferredAddress {
    ip_v4:IPAddress,
    ip_v6:IPAddress,

    port_v4:uint16,
    port_v6:uint16,

    connection_id:bytes,
    stateless_reset_token:Token
}
```

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.

5.3.4. 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.

Data:

```
{
    disable_active_migration?:boolean,

    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,
}
```

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

5.3.5. packet_sent

Importance: Core

Data:

```
{
    header:PacketHeader,

    frames?:Array<QuicFrame>, // see appendix for the definitions

    is_coalesced?:boolean, // default value is false

    retry_token?:Token, // only if header.packet_type === retry

    stateless_reset_token?:bytes, // only if header.packet_type === stateless_res
et. Is always 128 bits in length.

    supported_versions:Array<bytes>, // only if header.packet_type === version_ne
gotiation

    raw?:RawInfo,
    datagram_id?:uint32
}
```

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

Triggers:

* "retransmit_reordered" // draft-23 5.1.1

* "retransmit_timeout" // draft-23 5.1.2

* "pto_probe" // draft-23 5.3.1

* "retransmit_crypto" // draft-19 6.2

* "cc_bandwidth_probe" // needed for some CCs to figure out bandwidth allocations when there are no normal sends

Note: for more details on "datagram_id", see Section 5.3.10. It is only needed when keeping track of packet coalescing.

5.3.6. packet_received

Importance: Core

Data:

```
{
    header:PacketHeader,

    frames?:Array<QuicFrame>, // see appendix for the definitions

    is_coalesced?:boolean,

    retry_token?:Token, // only if header.packet_type === retry

    stateless_reset_token?:bytes, // only if header.packet_type === stateless_res
et. Is always 128 bits in length.

    supported_versions:Array<bytes>, // only if header.packet_type === version_ne
gotiation

    raw?:RawInfo,
    datagram_id?:uint32
}
```

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

Triggers:

* "keys_available" // if packet was buffered because it couldn't be decrypted before

Note: for more details on "datagram_id", see Section 5.3.10. It is only needed when keeping track of packet coalescing.

5.3.7. packet_dropped

Importance: Base

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

Data:

```
{
    header?:PacketHeader, // primarily packet_type should be filled here, as othe
r fields might not be parseable

    raw?:RawInfo,
    datagram_id?:uint32
}
```

For this event, the "trigger" field SHOULD be set (for example to one of the values below), as this helps tremendously in debugging.

Triggers:

* "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"

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 3.

Note: for more details on "datagram_id", see Section 5.3.10. It is only needed when keeping track of packet coalescing.

5.3.8. 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.

Data:

```
{
    header?:PacketHeader, // primarily packet_type and possible packet_number sho
uld be filled here, as other elements might not be available yet

    raw?:RawInfo,
    datagram_id?:uint32
}
```

Note: for more details on "datagram_id", see Section 5.3.10. It is only needed when keeping track of packet coalescing.

Triggers:

* "backpressure" // indicates the parser cannot keep up, temporarily buffers packet for later processing

* "keys_unavailable" // if packet cannot be decrypted because the proper keys were not yet available

5.3.9. 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.

Data: ~~~ { packet_number_space?:PacketNumberSpace,

packet_numbers?:Array<uint64> } ~~~

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.

5.3.10. 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.

Data:

```
{
    count?:uint16, // to support passing multiple at once
    raw?:Array<RawInfo>, // RawInfo:length field indicates total length of the da
tagrams, including UDP header length

    datagram_ids?:Array<uint32>
}
```

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.

5.3.11. 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.

Data:

```
{
    count?:uint16, // to support passing multiple at once
    raw?:Array<RawInfo>, // RawInfo:length field indicates total length of the da
tagrams, including UDP header length

    datagram_ids?:Array<uint32>
}
```

Note: for more details on "datagram_ids", see Section 5.3.10.

5.3.12. 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).

Data:

```
{
    raw?:RawInfo
}
```

5.3.13. 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.

Data:

```
{
    stream_id:uint64,
    stream_type?:"unidirectional"|"bidirectional", // mainly useful when opening
the stream

    old?:StreamState,
    new:StreamState,

    stream_side?:"sending"|"receiving"
}

enum 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
    destroyed // memory actually freed
}
```

   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.

5.3.14. 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 (contrary to the HTTP-level "frames_parsed" 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.

Data:

```
{
    frames:Array<QuicFrame>, // see appendix for the definitions

    packet_number?:uint64
}
```

5.3.15. data_moved

Importance: Base

Used to indicate when data moves between the different layers (for example passing from HTTP/3 to QUIC stream buffers and vice versa) or between HTTP/3 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 HTTP layer immediately (for example per received packet) or in larger batches (for example, all QUIC packets are processed first and afterwards the HTTP layer reads from the streams with newly available data). This in turn can help identify bottlenecks or scheduling problems.

Data:

```
{
    stream_id?:uint64,
    offset?:uint64,
    length?:uint64, // byte length of the moved data

    from?:string, // typically: use either of "application","http","transport"
    to?:string, // typically: use either of "application","http","transport"

    data?:bytes // raw bytes that were transferred
}
```

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.

5.4. 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).

5.4.1. 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.

Data:

```
{
    // Loss detection, see recovery draft-23, Appendix A.2
    reordering_threshold?:uint16, // in amount of packets
    time_threshold?:float, // as RTT multiplier
    timer_granularity?:uint16, // in ms
    initial_rtt?:float, // in ms

    // congestion control, Appendix B.1.
    max_datagram_size?:uint32, // in bytes // Note: this could be updated after p
mtud
    initial_congestion_window?:uint64, // in bytes
    minimum_congestion_window?:uint32, // in bytes // Note: this could change whe
n max_datagram_size changes
    loss_reduction_factor?:float,
    persistent_congestion_threshold?:uint16 // as PTO multiplier
}
```

Additionally, this event can contain any number of unspecified fields to support different recovery approaches.

5.4.2. 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.

Data:

```
{
    // Loss detection, see recovery draft-23, Appendix A.3
    min_rtt?:float, // in ms or us, depending on the overarching qlog's configura
tion
    smoothed_rtt?:float, // in ms or us, depending on the overarching qlog's conf
iguration
    latest_rtt?:float, // in ms or us, depending on the overarching qlog's config
uration
    rtt_variance?:float, // in ms or us, depending on the overarching qlog's conf
iguration

    pto_count?:uint16,

    // Congestion control, Appendix B.2.
    congestion_window?:uint64, // in bytes
    bytes_in_flight?:uint64,

    ssthresh?:uint64, // in bytes

    // qlog defined
    packets_in_flight?:uint64, // sum of all packet number spaces

    pacing_rate?:uint64 // in bps
}
```

Note: to make logging easier, implementations MAY log values even if they are the same as previously reported values (e.g., two subsequent METRIC_UPDATE 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.

5.4.3. 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

Data:

```
{
    old?:string,
    new:string
}
```

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 occuring (e.g., slow start is exited only when ssthresh is exceeded).

Some triggers for ("enhanced" New Reno):

* persistent_congestion
* ECN

5.4.4. 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.

Data:

```
{
    timer_type?:"ack"|"pto", // called "mode" in draft-23 A.9.
    packet_number_space?: PacketNumberSpace,

    event_type:"set"|"expired"|"cancelled",

    delta?:float // if event_type === "set": delta time in ms or us (see configur
ation) from this event's timestamp until when the timer will trigger
}
```

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

5.4.5. packet_lost

Importance: Core

This event is emitted when a packet is deemed lost by loss detection.

Data:

```
{
    header?:PacketHeader, // should include at least the packet_type and packet_n
umber

    // not all implementations will keep track of full packets, so these are opti
onal
    frames?:Array<QuicFrame> // see appendix for the definitions
}
```

For this event, the "trigger" field SHOULD be set (for example to one of the values below), as this helps tremendously in debugging.

Triggers:

* "reordering_threshold",

* "time_threshold"

* "pto_expired" // draft-23 section 5.3.1, MAY

5.4.6. marked_for_retransmit

Importance: Extra

This event indicates which data was marked for retransmit upon detecing 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 consituent 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).

Data:

```
{
    frames:Array<QuicFrame>, // see appendix for the definitions
}
```

6. HTTP/3 event definitions

6.1. http

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

6.1.1. 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.

Data:

```
{
    owner?:"local" | "remote",

    max_header_list_size?:uint64, // from SETTINGS_MAX_HEADER_LIST_SIZE
    max_table_capacity?:uint64, // from SETTINGS_QPACK_MAX_TABLE_CAPACITY
    blocked_streams_count?:uint64, // from SETTINGS_QPACK_BLOCKED_STREAMS

    // qlog-defined
    waits_for_settings?:boolean // indicates whether this implementation waits fo
r a SETTINGS frame before processing requests
}
```

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.

6.1.2. parameters_restored

Importance: Base

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

Data:

```
{
    max_header_list_size?:uint64,
    max_table_capacity?:uint64,
    blocked_streams_count?:uint64
}
```

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

6.1.3. 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.

Data:

```
{
    stream_id:uint64,

    owner?:"local"|"remote"

    old?:StreamType,
    new:StreamType,

    associated_push_id?:uint64 // only when new == "push"
}

enum StreamType {
    data, // bidirectional request-response streams
    control,
    push,
    reserved,
    qpack_encode,
    qpack_decode
}
```

6.1.4. 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.

Data:

```
{
    stream_id:uint64,
    length?:uint64, // payload byte length of the frame
    frame:HTTP3Frame, // see appendix for the definitions,

    raw?:RawInfo
}
```

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.

6.1.5. 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.

Data:

```
{
    stream_id:uint64,
    length?:uint64, // payload byte length of the frame
    frame:HTTP3Frame, // see appendix for the definitions,

    raw?:RawInfo
}
```

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.

6.1.6. 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.

```
{
    push_id?:uint64,
    stream_id?:uint64, // in case this is logged from a place that does not have
access to the push_id

    decision:"claimed"|"abandoned"
}
```

6.2. 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.

6.2.1. 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.

Data:

```
{
    owner:"local" | "remote",

    dynamic_table_capacity?:uint64,
    dynamic_table_size?:uint64, // effective current size, sum of all the entries

    known_received_count?:uint64,
    current_insert_count?:uint64
}
```

6.2.2. 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.

Data:

```
{
    stream_id:uint64,

    state:"blocked"|"unblocked" // streams are assumed to start "unblocked" until
 they become "blocked"
}
```

6.2.3. dynamic_table_updated

Importance: Extra

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

Data:

```
{
    owner:"local" | "remote", // local = the encoder's dynamic table. remote = th
e decoder's dynamic table

    update_type:"inserted"|"evicted",

    entries:Array<DynamicTableEntry>
}

class DynamicTableEntry {
    index:uint64;
    name?:string | bytes;
    value?:string | bytes;
}
```

6.2.4. 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.

Data:

```
{
    stream_id?:uint64,

    headers?:Array<HTTPHeader>,

    block_prefix:QPackHeaderBlockPrefix,
    header_block:Array<QPackHeaderBlockRepresentation>,

    length?:uint32,
    raw?:bytes
}
```

6.2.5. 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.

Data:

```
{
    stream_id?:uint64,

    headers?:Array<HTTPHeader>,

    block_prefix:QPackHeaderBlockPrefix,
    header_block:Array<QPackHeaderBlockRepresentation>,

    length?:uint32,
    raw?:bytes
}
```

6.2.6. 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.

Data:

```
{
    instruction:QPackInstruction // see appendix for the definitions,

    length?:uint32,
    raw?:bytes
}
```

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)

6.2.7. instruction_parsed

Importance: Base

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

Data:

```
{
    instruction:QPackInstruction // see appendix for the definitions,

    length?:uint32,
    raw?:bytes
}
```

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)

7. Generic events and Simulation indicators

7.1. generic

The main goal of the events in this category is to allow implementations to fully replace their existing text-based logging by qlog. This is done by providing events to log generic strings for typical well-known logging levels (error, warning, info, debug, verbose).

7.1.1. error

Importance: Core

Used to log details of an internal error. For errors that effectively lead to the closure of a QUIC connection, it is recommended to use transport:connection_closed instead.

Data:

```
{
    code?:uint32,
    message?:string
}
```

7.1.2. warning

Importance: Base

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

Data:

```
{
    code?:uint32,
    message?:string
}
```

7.1.3. 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.

Data:

```
{
    message:string
}
```

7.1.4. 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.

Data:

```
{
    message:string
}
```

7.1.5. 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.

Data:

```
{
    message:string
}
```

7.2. simulation

When evaluating a protocol evaluation, 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.

7.2.1. 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.

```
{
    name?:string,
    details?:any
}
```

7.2.2. 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).

```
{
    type?:string,
    message?:string
}
```

8. Security Considerations

TBD

9. IANA Considerations

TBD

10. References

10.1. Normative References

[QLOG-MAIN]
Marx, R., Ed., "Main logging schema for qlog", Work in Progress, Internet-Draft, draft-marx-qlog-main-schema-02, 2 November 2020, <https://tools.ietf.org/html/draft-marx-qlog-main-schema-02>.

[QUIC-HTTP]
Bishop, M., Ed., "Hypertext Transfer Protocol Version 3 (HTTP/3)", Work in Progress, Internet-Draft, draft-ietf-quic-http-32, 1 October 2020, <https://tools.ietf.org/html/draft-ietf-quic-http-32>.

[QUIC-QPACK]
Frindell, A., Ed., "QPACK: Header Compression for HTTP/3", Work in Progress, Internet-Draft, draft-ietf-quic-qpack-19, 20 October 2020, <https://tools.ietf.org/html/draft-ietf-quic-qpack-19>.

[QUIC-TRANSPORT]
Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", Work in Progress, Internet-Draft, draft-ietf-quic-transport-32, 1 October 2020, <https://tools.ietf.org/html/draft-ietf-quic-transport-32>.

10.2. 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/info/rfc2119>.

Appendix A. QUIC data field definitions

A.1. IPAddress

```
class IPAddress : string | bytes;

// an IPAddress can either be a "human readable" form (e.g., "127.0.0.1" for v4 o
r "2001:0db8:85a3:0000:0000:8a2e:0370:7334" for v6) or use a raw byte-form (as th
e string forms can be ambiguous)
```

A.2. PacketType

```
enum PacketType {
    initial,
    handshake,
    zerortt = "0RTT",
    onertt = "1RTT",
    retry,
    version_negotiation,
    stateless_reset,
    unknown
}
```

A.3. PacketNumberSpace

```
enum PacketNumberSpace {
    initial,
    handshake,
    application_data
}
```

A.4. PacketHeader

```
class PacketHeader {
    // Note: short vs long header is implicit through PacketType

    packet_type: PacketType;
    packet_number: uint64;

    flags?: uint8; // the bit flags of the packet headers (spin bit, key update b
it, etc. up to and including the packet number length bits if present) interprete
d as a single 8-bit integer

    token?:Token; // only if packet_type == initial

    length?: uint16, // only if packet_type == initial || handshake || 0RTT. Sign
ifies length of the packet_number plus the payload.

    // only if present in the header
    // if correctly using transport:connection_id_updated events,
    // dcid can be skipped for 1RTT packets
    version?: bytes; // e.g., "ff00001d" for draft-29
    scil?: uint8;
    dcil?: uint8;
    scid?: bytes;
    dcid?: bytes;
}
```

A.5. Token

```
class Token {
    type?:"retry"|"resumption"|"stateless_reset";

    length?:uint32; // byte length of the token
    data?:bytes; // raw byte value of the token

    details?:any; // decoded fields included in the token (typically: peer's IP a
ddress, creation time)
}
```

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.6. KeyType

```
   enum 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
   }
```

A.7. QUIC Frames

```
type QuicFrame = PaddingFrame | PingFrame | AckFrame | ResetStreamFrame | StopSen
dingFrame | CryptoFrame | NewTokenFrame | StreamFrame | MaxDataFrame | MaxStreamD
ataFrame | MaxStreamsFrame | DataBlockedFrame | StreamDataBlockedFrame | StreamsB
lockedFrame | NewConnectionIDFrame | RetireConnectionIDFrame | PathChallengeFrame
 | PathResponseFrame | ConnectionCloseFrame | HandshakeDoneFrame | UnknownFrame;
```

A.7.1. 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.

```
   class PaddingFrame{
       frame_type:string = "padding";

       length?:uint32; // total frame length, including frame header
       payload_length?:uint32;
   }
```

A.7.2. PingFrame

```
   class PingFrame{
       frame_type:string = "ping";

       length?:uint32; // total frame length, including frame header
       payload_length?:uint32;
   }
```

A.7.3. AckFrame

```
class AckFrame{
    frame_type:string = "ack";

    ack_delay?:float; // in ms

    // first number is "from": lowest packet number in interval
    // second number is "to": up to and including // highest packet number in int
erval
    // e.g., looks like [[1,2],[4,5]]
    acked_ranges?:Array<[uint64, uint64]|[uint64]>;

    // ECN (explicit congestion notification) related fields (not always present)
    ect1?:uint64;
    ect0?:uint64;
    ce?:uint64;

    length?:uint32; // total frame length, including frame header
    payload_length?:uint32;
}
```

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.7.4. ResetStreamFrame

```
class ResetStreamFrame{
    frame_type:string = "reset_stream";

    stream_id:uint64;
    error_code:ApplicationError | uint32;
    final_size:uint64; // in bytes

    length?:uint32; // total frame length, including frame header
    payload_length?:uint32;
}
```

A.7.5. StopSendingFrame

```
   class StopSendingFrame{
       frame_type:string = "stop_sending";

       stream_id:uint64;
       error_code:ApplicationError | uint32;

       length?:uint32; // total frame length, including frame header
       payload_length?:uint32;
   }
```

A.7.6. CryptoFrame

```
   class CryptoFrame{
       frame_type:string = "crypto";

       offset:uint64;
       length:uint64;

       payload_length?:uint32;
   }
```

A.7.7. NewTokenFrame

```
   class NewTokenFrame{
     frame_type:string = "new_token";

     token:Token
   }
```

A.7.8. StreamFrame

```
class StreamFrame{
    frame_type:string = "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?:boolean;

    raw?:bytes;
}
```

A.7.9. MaxDataFrame

```
class MaxDataFrame{
  frame_type:string = "max_data";

  maximum:uint64;
}
```

A.7.10. MaxStreamDataFrame

```
class MaxStreamDataFrame{
  frame_type:string = "max_stream_data";

  stream_id:uint64;
  maximum:uint64;
}
```

A.7.11. MaxStreamsFrame

```
class MaxStreamsFrame{
  frame_type:string = "max_streams";

  stream_type:string = "bidirectional" | "unidirectional";
  maximum:uint64;
}
```

A.7.12. DataBlockedFrame

```
class DataBlockedFrame{
  frame_type:string = "data_blocked";

  limit:uint64;
}
```

A.7.13. StreamDataBlockedFrame

```
class StreamDataBlockedFrame{
  frame_type:string = "stream_data_blocked";

  stream_id:uint64;
  limit:uint64;
}
```

A.7.14. StreamsBlockedFrame

```
   class StreamsBlockedFrame{
     frame_type:string = "streams_blocked";

     stream_type:string = "bidirectional" | "unidirectional";
     limit:uint64;
   }
```

A.7.15. NewConnectionIDFrame

```
   class NewConnectionIDFrame{
     frame_type:string = "new_connection_id";

     sequence_number:uint32;
     retire_prior_to:uint32;

     connection_id_length?:uint8;
     connection_id:bytes;

     stateless_reset_token?:Token;
   }
```

A.7.16. RetireConnectionIDFrame

```
   class RetireConnectionIDFrame{
     frame_type:string = "retire_connection_id";

     sequence_number:uint32;
   }
```

A.7.17. PathChallengeFrame

```
   class PathChallengeFrame{
     frame_type:string = "path_challenge";

     data?:bytes; // always 64-bit
   }
```

A.7.18. PathResponseFrame

```
   class PathResponseFrame{
     frame_type:string = "path_response";

     data?:bytes; // always 64-bit
   }
```

A.7.19. 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).

```
type ErrorSpace = "transport" | "application";

class ConnectionCloseFrame{
    frame_type:string = "connection_close";

    error_space?:ErrorSpace;
    error_code?:TransportError | ApplicationError | uint32;
    raw_error_code?:uint32;
    reason?:string;

    trigger_frame_type?:uint64 | string; // For known frame types, the appropriat
e "frame_type" string. For unknown frame types, the hex encoded identifier value
}
```

A.7.20. HandshakeDoneFrame

```
class HandshakeDoneFrame{
  frame_type:string = "handshake_done";
}
```

A.7.21. UnknownFrame

```
class UnknownFrame{
    frame_type:string = "unknown";
    raw_frame_type:uint64;

    raw_length?:uint32;
    raw?:bytes;
}
```

A.7.22. TransportError

```
   enum 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
   }
```

A.7.23. 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 value of the TLS alert description.

```
   enum CryptoError {
       crypto_error_{TLS_ALERT}
   }
```

Appendix B. HTTP/3 data field definitions

B.1. HTTP/3 Frames

```
type HTTP3Frame = DataFrame | HeadersFrame | PriorityFrame | CancelPushFrame | Se
ttingsFrame | PushPromiseFrame | GoAwayFrame | MaxPushIDFrame | DuplicatePushFram
e | ReservedFrame | UnknownFrame;
```

B.1.1. DataFrame

```
   class DataFrame{
       frame_type:string = "data";

       raw?:bytes;
   }
```

B.1.2. 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"}]
```

```
class HeadersFrame{
    frame_type:string = "header";
    headers:Array<HTTPHeader>;
}

class HTTPHeader {
    name:string;
    value:string;
}
```

B.1.3. CancelPushFrame

```
class CancelPushFrame{
    frame_type:string = "cancel_push";
    push_id:uint64;
}
```

B.1.4. SettingsFrame

```
class SettingsFrame{
    frame_type:string = "settings";
    settings:Array<Setting>;
}

class Setting{
    name:string;
    value:string;
}
```

B.1.5. PushPromiseFrame

```
class PushPromiseFrame{
    frame_type:string = "push_promise";
    push_id:uint64;

    headers:Array<HTTPHeader>;
}
```

B.1.6. GoAwayFrame

```
class GoAwayFrame{
    frame_type:string = "goaway";
    stream_id:uint64;
}
```

B.1.7. MaxPushIDFrame

```
class MaxPushIDFrame{
    frame_type:string = "max_push_id";
    push_id:uint64;
}
```

B.1.8. DuplicatePushFrame

```
class DuplicatePushFrame{
    frame_type:string = "duplicate_push";
    push_id:uint64;
}
```

B.1.9. ReservedFrame

```
class ReservedFrame{
    frame_type:string = "reserved";
}
```

B.1.10. UnknownFrame

HTTP/3 re-uses QUIC's UnknownFrame definition, since their values and usage overlaps.

B.2. ApplicationError

```
   enum ApplicationError{
       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
   }
```

Appendix C. QPACK DATA type definitions

C.1. QPACK Instructions

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

```
type QPackInstruction = SetDynamicTableCapacityInstruction | InsertWithNameRefere
nceInstruction | InsertWithoutNameReferenceInstruction | DuplicateInstruction | H
eaderAcknowledgementInstruction | StreamCancellationInstruction | InsertCountIncr
ementInstruction;
```

C.1.1. SetDynamicTableCapacityInstruction

```
   class SetDynamicTableCapacityInstruction {
       instruction_type:string = "set_dynamic_table_capacity";

       capacity:uint32;
   }
```

C.1.2. InsertWithNameReferenceInstruction

```
   class InsertWithNameReferenceInstruction {
       instruction_type:string = "insert_with_name_reference";

       table_type:"static"|"dynamic";

       name_index:uint32;

       huffman_encoded_value:boolean;

       value_length?:uint32;
       value?:string;
   }
```

C.1.3. InsertWithoutNameReferenceInstruction

```
   class InsertWithoutNameReferenceInstruction {
       instruction_type:string = "insert_without_name_reference";

       huffman_encoded_name:boolean;

       name_length?:uint32;
       name?:string;

       huffman_encoded_value:boolean;

       value_length?:uint32;
       value?:string;
   }
```

C.1.4. DuplicateInstruction

```
   class DuplicateInstruction {
       instruction_type:string = "duplicate";

       index:uint32;
   }
```

C.1.5. HeaderAcknowledgementInstruction

```
   class HeaderAcknowledgementInstruction {
       instruction_type:string = "header_acknowledgement";

       stream_id:uint64;
   }
```

C.1.6. StreamCancellationInstruction

```
   class StreamCancellationInstruction {
       instruction_type:string = "stream_cancellation";

       stream_id:uint64;
   }
```

C.1.7. InsertCountIncrementInstruction

```
   class InsertCountIncrementInstruction {
       instruction_type:string = "insert_count_increment";

       increment:uint32;
   }
```

C.2. QPACK Header compression

```
type QPackHeaderBlockRepresentation = IndexedHeaderField | LiteralHeaderFieldWith
Name | LiteralHeaderFieldWithoutName;
```

C.2.1. IndexedHeaderField

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

```
class IndexedHeaderField {
    header_field_type:string = "indexed_header";

    table_type:"static"|"dynamic"; // MUST be "dynamic" if is_post_base is true
    index:uint32;

    is_post_base:boolean = false; // to represent the "indexed header field with
post-base index" header field type
}
```

C.2.2. LiteralHeaderFieldWithName

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

```
class LiteralHeaderFieldWithName {
    header_field_type:string = "literal_with_name";

    preserve_literal:boolean; // the 3rd "N" bit
    table_type:"static"|"dynamic"; // MUST be "dynamic" if is_post_base is true
    name_index:uint32;

    huffman_encoded_value:boolean;
    value_length?:uint32;
    value?:string;

    is_post_base:boolean = false; // to represent the "Literal header field with
post-base name reference" header field type
}
```

C.2.3. LiteralHeaderFieldWithoutName

```
class LiteralHeaderFieldWithoutName {
    header_field_type:string = "literal_without_name";

    preserve_literal:boolean; // the 3rd "N" bit

    huffman_encoded_name:boolean;
    name_length?:uint32;
    name?:string;

    huffman_encoded_value:boolean;
    value_length?:uint32;
    value?:string;
}
```

C.2.4. QPackHeaderBlockPrefix

```
class QPackHeaderBlockPrefix {
    required_insert_count:uint32;
    sign_bit:boolean;
    delta_base:uint32;
}
```

Appendix D. Change Log

D.1. Since draft-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)

D.2. Since draft-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 E. Design Variations

TBD

Appendix F. Acknowledgements

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.

Author's Address

Robin Marx
Hasselt University

Email: robin.marx@uhasselt.be

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

R. Marx
Hasselt University

2 November 2020

Main logging schema for qlog
draft-marx-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 6 May 2021.

Copyright Notice

Table of Contents

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-HTTP3] for QUIC and HTTP/3-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 welcome at https://github.com/quiclog/internet-drafts (https://github.com/quiclog/internet-drafts). Readers are advised to refer to the "editor's draft" at that URL for an up-to-date version of this document.

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].

While the qlog schema's are format-agnostic, for readability the qlog documents will use a JSON-inspired format ([RFC8259]) for examples and definitions.

As qlog can be serialized both textually but also in binary, we employ a custom datatype definition language, inspired loosely by the "TypeScript" language (https://www.typescriptlang.org/).

This document describes how to employ JSON and NDJSON as textual serializations for qlog in Section 4. Other documents will describe how to utilize other concrete serialization options, though tips and requirements for these are also listed in this document (Section 4).

The main general conventions in this document a reader should be aware of are:

* obj? : this object is optional
* type1 | type2 : a union of these two types (object can be either type1 OR type2)
* obj:type : this object has this concrete type
* obj:array<type> : this object is an array of this type
* class : defines a new type
* // : single-line comment

The main data types are:

* int8 : signed 8-bit integer
* int16 : signed 16-bit integer
* int32 : signed 32-bit integer
* int64 : signed 64-bit integer
* uint8 : unsigned 8-bit integer
* uint16 : unsigned 16-bit integer
* uint32 : unsigned 32-bit integer
* uint64 : unsigned 64-bit integer
* float : 32-bit floating point value

* double : 64-bit floating point value

* byte : an individual raw byte (8-bit) value (use array<byte> or the shorthand "bytes" to specify a binary blob)

* string : list of Unicode (typically UTF-8) encoded characters

* boolean : boolean

* enum: fixed list of values (Unless explicity defined, the value of an enum entry is the string version of its name (e.g., initial = "initial"))

* any : represents any object type. Mainly used here as a placeholder for more concrete types defined in related documents (e.g., specific event types)

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

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

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 (e.g., no complex fixed schema with extension points)

* Aggregation and transformation friendly (e.g., the top-level element is a container for individual traces, group_id 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 indivdual 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 "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 4) 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 1.

Definition:

```
class QlogFile {
    qlog_version:string,
    qlog_format?:string,
    title?:string,
    description?:string,
    summary?: Summary,
    traces: array<Trace|TraceError>
}
```

JSON serialization:

```
{
    "qlog_version": "draft-02",
    "qlog_format": "JSON",
    "title": "Name of this particular qlog file (short)",
    "description": "Description for this group of traces (long)",
    "summary": {
        ...
    },
    "traces": [...]
}
```

Figure 1: Top-level element

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 Figure 2.

Definition (purely illustrative example):

```
class Summary {
    "trace_count":uint32, // amount of traces in this file
    "max_duration":uint64, // time duration of the longest trace in ms
    "max_outgoing_loss_rate":float, // highest loss rate for outgoing packets ove
r all traces
    "total_event_count":uint64, // total number of events across all traces,
    "error_count":uint64 // total number of error events in this trace
}
```

JSON serialization:

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

Figure 2: Summary example definition

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 3.

Definition:

```
class TraceError {
    error_description: string, // A description of the error
    uri?: string, // the original URI at which we attempted to find the file
    vantage_point?: VantagePoint // see {{vantage_point}}: the vantage point we w
ere expecting to include here
}
```

JSON serialization:

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

Figure 3: TraceError definition

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 3.4.7.

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 4.

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:

```
class Trace {
    title?: string,
    description?: string,
    configuration?: Configuration,
    common_fields?: CommonFields,
    vantage_point: VantagePoint,
    events: array<Event>
}
```

JSON serialization:

```
{
    "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 4: Trace container definition

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:

```
class Configuration {
    time_offset:double, // in ms,
    original_uris: array<string>,

    // list of fields with any type
}
```

JSON serialization:

```
{
    "time_offset": 150, // starts 150ms after the first timestamp indicates
    "original_uris": [
        "https://example.org/trace1.qlog",
        "https://example.org/trace2.qlog"
    ]
}
```

Figure 5: Configuration definition

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 6.

```
{
    "configuration" : {
        "qvis" : {
            // when loaded into the qvis toolsuite's congestion graph tool
            // zoom in on the period between 1s and 2s and select the 124th event
 defined in this trace
            "congestion_graph": {
                "startX": 1000,
                "endX": 2000,
                "focusOnEventIndex": 124
            }


            // when loaded into the qvis toolsuite's sequence diagram tool
            // automatically scroll down the timeline to the 555th event defined
in this trace
            "sequence_diagram" : {
                "focusOnEventIndex": 555
            }
        }
    }
}
```

Figure 6: Custom configuration fields example

3.3.2. vantage_point

The vantage_point field describes the vantage point from which the trace originates, see Figure 7. 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.

Definition:

```
class VantagePoint {
    name?: string,
    type: VantagePointType,
    flow?: VantagePointType
}

class VantagePointType {
    server, // endpoint which initiates the connection.
    client, // endpoint which accepts the connection.
    network, // observer in between client and server.
    unknown
}
```

JSON serialization examples:

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

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

Figure 7: VantagePoint definition

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 3.4.6), and "group_id" Section 3.4.7. As especially these later fields typically have identical values across individual event instances, they are normally logged separately in the "common_fields" (Section 3.4.8).

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 and HTTP/3 can be found in [QLOG-QUIC-HTTP3].

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-HTTP3].

An example of a qlog event with its component fields is shown in Figure 8.

```
Definition:

class Event {
    time: double,
    name: string,
    data: any,

    protocol_type?: string,
    group_id?: string|uint32,

    time_format?: "absolute"|"delta"|"relative",

    // list of fields with any type
}

JSON serialization:

{
    time: 1553986553572,

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

    protocol_type:  "QUIC_HTTP3",
    group_id: "127ecc830d98f9d54a42c4f0842aa87e181a",

    time_format: "absolute",

    ODCID: "127ecc830d98f9d54a42c4f0842aa87e181a", // QUIC specific
}
```

Figure 8: Event fields definition

3.4.1. timestamps

The "time" field indicates the timestamp at which the event occured. 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 succint 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":

* 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 3.4.8) 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 9.

```
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 9: 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 postprocesser 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

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 10. 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 10: 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 definitons for QUIC and HTTP/3, see [QLOG-QUIC-HTTP3].

One purely illustrative example for a QUIC "packet_sent" event is shown in Figure 11.

Definition:

```
class TransportPacketSentEvent {
    packet_size?:uint32,
    header:PacketHeader,
    frames?:Array<QuicFrame>
}
```

JSON serialization:

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

Figure 11: Example of the 'data' field for a QUIC packet_sent event

3.4.4. protocol_type

The "protocol_type" field indicates to which protocol (or protocol "stack") 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).

For example, QUIC and HTTP/3 events have the "QUIC_HTTP3" protocol_type value, see [QLOG-QUIC-HTTP3].

Typically however, all events in a single trace are of the same protocol, and this field is logged once in "common_fields", see Section 3.4.8.

3.4.5. custom fields

Note that qlog files can always contain custom fields (e.g., a per-event field indicating its privacy properties or path_id in multipath protocols) and assign custom values to existing fields (e.g., new categories/types for implemenation-specific events). Loggers are free to add such fields and field values and tools MUST either ignore these unknown fields or show them in a generic fashion.

3.4.6. 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 occuring. 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 12.

Definition:

```
class QuicPacketDroppedEvent {
    packet_type?:PacketType,
    raw_length?:uint32,

    trigger?: "key_unavailable" | "unknown_connection_id" | "decrypt_error" | "un
supported_version"
}
```

Figure 12: Trigger example

3.4.7. 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 13.

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

```
events: [
    {
        time: 1553986553579,
        protocol_type: "TCP_HTTPS2",
        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 13: Example of group_id usage

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 3.4.8.

3.4.8. 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 14.

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 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 14: Example of common_fields usage

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 uknown 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:

* time_format
* reference_time
* protocol_type
* group_id

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. 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 both how to practically transform qlog schema definitions to JSON and to the streamable NDJSON. We discuss concrete options to bring down JSON size and processing overheads in Section 4.3.

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 4.3).

4.1. 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.

To facilitate this mapping, the qlog documents employ a format that is close to pure JSON for its examples and data definitions. Still, as JSON is not a typed format, there are some practical peculiarities to observe.

4.1.1. numbers

While JSON has built-in support for integers up to 64 bits in size, not all JSON parsers do. For example, none of the major Web browsers support full 64-bit integers at this time, as all numerical values (both floating-point numbers and integers) are internally represented as floating point IEEE 754 (https://en.wikipedia.org/wiki/Floating-point_arithmetic) values. In practice, this limits their integers to a maximum value of 2^53-1. Integers larger than that are either truncated or produce a JSON parsing error. While this is expected to improve in the future (as "BigInt" support (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/BigInt) has been introduced in most Browsers, though not yet integrated into JSON parsers), we still need to deal with it here.

When transforming an int64, uint64 or double from qlog to JSON, the implementer can thus choose to either log them as JSON numbers (taking the risk of truncation or un-parseability) or to log them as strings instead. Logging as strings should however only be

practically needed if the value is likely to exceed 2^53-1. In practice, even though protocols such as QUIC allow 64-bit values for for example stream identifiers, these high numbers are unlikely to be reached for the overwhelming majority of cases. As such, it is probably a valid trade-off to take the risk and log 64-bit values as JSON numbers instead of strings.

Tools processing JSON-based qlog SHOULD however be able to deal with 64-bit fields being serialized as either strings or numbers.

4.1.2. bytes

Unlike most binary formats, JSON does not allow the logging of raw binary blobs directly. As such, when serializing a byte or array<byte>, a scheme needs to be chosen.

To represent qlog bytes in JSON, they MUST be serialized to their lowercase hexadecimal equivalents (with 0 prefix for values lower than 10). All values are directly appended to each other, without delimiters. The full value is not prefixed with 0x (as is sometimes common). An example is given in Figure 15.

For the five raw unsigned byte input values of: 5 20 40 171 255, the JSON seriali
zation is:

```
{
    raw: "051428abff"
}
```

Figure 15: Example for serializing bytes

As such, the resulting string will always have an even amount of characters and the original byte-size can be retrieved by dividing the string length by 2.

4.1.2.1. Truncated values

In some cases, it can be interesting not to log a full raw blob 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 array<byte> they specify. This allows always retrieving the original length, 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. All possible permutations are shown by example in Figure 16.

```
// 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 16: Example for serializing truncated bytes

4.1.3. Summarizing table

By definition, JSON strings are serialized surrounded by quotes. Numbers without.

qlog type	JSON type
int8	number
int16	number
int32	number
uint8	number
uint16	number
uint32	number
float	number
int64	number or string
uint64	number or string
double	number or string
bytes	string (lowercase hex value)
string	string
boolean	string ("true" or "false")
enum	string (full value/name, not index)
any	object ({...})
array	array ([...])

Table 1

4.1.4. Other JSON specifics

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

Most JSON parsers strictly follow the JSON specification. This includes the rule that trailing comma's are not allowed. As it is frequently annoying to remove these trailing comma's when logging events in a streaming fashion, tool implementers SHOULD allow the last event entry of a qlog trace to be an empty object. This allows loggers to simply close the qlog file by appending "{}]}]}" after their last added event.

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

4.2. qlog to NDJSON 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 Newline-Delimited JSON (NDJSON) (http://ndjson.org/).

When mapping qlog to NDJSON, the "qlog_format" field MUST have the value "NDJSON".

NDJSON is very similar to JSON, except that it interprets each line in a file as a fully separate JSON object. Put differently, unlike default JSON, it does not require a file to be wrapped as a full object with "{ ... }" or "[...]". Using this setup, qlog events can simply be appended as individually serialized lines at the back of a streamed logging file.

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 "file header" (QlogFile class in Section 3). Additionally, qlog's NDJSON mapping does not allow logging multiple individual traces in a single qlog file. As such, the QlogFile:traces field is replaced by the singular "trace" field, which simply contains the Trace data directly. An example can be seen in Figure 17. 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 NDJSON qlog file.

Note as well from Figure 17 that the file's header (QlogFileNDJSON) also needs to be fully serialized on a single line to be NDJSON compatible.

Definition:

```
class QlogFileNDJSON {
    qlog_format: "NDJSON",

    qlog_version:string,
    title?:string,
    description?:string,
    summary?: Summary,
    trace: Trace
}
// list of qlog events, separated by newlines
```

NDJSON serialization:

```
{"qlog_format":"NDJSON","qlog_version":"draft-02","title":"Name of this particula
r NDJSON qlog file (short)","description":"Description for this NDJSON qlog file
(long)","trace":{"common_fields":{"protocol_type":"QUIC_HTTP3","group_id":"127ecc
830d98f9d54a42c4f0842aa87e181a","time_format":"relative","reference_time":"155398
6553572"},"vantage_point":{"name":"backend-67","type":"server"}}}
{"time": 2, "name": "transport:packet_received", "data": { ... } }
{"time": 7, "name": "http:frame_parsed", "data": { ... } }
```

Figure 17: Top-level element

Finally, while not specifically required by the NDJSON specification, all qlog field names in a NDJSON serialization MUST be lowercase.

4.2.1. Supporting NDJSON in tooling

Note that NDJSON is not supported in most default programming environments (unlike normal JSON). However, several custom NDJSON parsing libraries exist (http://ndjson.org/libraries.html) that can be used and the format is easy enough to parse with existing implementations (i.e., by splitting the file into its component lines and feeding them to a normal JSON parser individually, as each line by itself is a valid JSON object).

4.3. Other optimizated formatting options

Both the JSON and NDJSON 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 (ND)JSON (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 4.3.2, 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 6), 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 ".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.

4.3.1. Data structure optimizations

The first general category of optimizations is to alter the representation of data within an (ND)JSON 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 "NDJSON.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 "NDJSON.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 4.3.2), these schemes are not inherently part of qlog.

4.3.2. Compression

The second general category of optimizations is to utilize a (generic) compression scheme for textual data. As qlog in the (ND)JSON 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 .qlog.gz SHOULD BE used. The "qlog_format" field should still reflect the original JSON formatting of the qlog data (e.g., "JSON" or "NDJSON").

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 .qlog.br SHOULD BE used. The "qlog_format" field should still reflect the original JSON formatting of the qlog data (e.g., "JSON" or "NDJSON").

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 6).

4.3.3. 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 straighforward 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 NDJSON qlogs). While CBOR thus does retain the full qlog flexibility, it only provides a 25% file size reduction (100MB to 75MB) compared to textual (ND)JSON. 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 .qlog.cbor SHOULD BE used. The "qlog_format" field should still reflect the original JSON formatting of the qlog data (e.g., "JSON" or "NDJSON").

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 it 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 .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 4.3.2). 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 4.3.2). 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.

4.3.4. 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
JSON Section 4.1	JSON	.qlog
NDJSON Section 4.2	NDJSON	.qlog
named headers Section 4.3.1	(ND)JSON.namedheaders	.qlog
dictionary Section 4.3.1	(ND)JSON.dictionary	.qlog
CBOR Section 4.3.3	(ND)JSON	.qlog.cbor
protobuf Section 4.3.3	protobuf	.qlog.protobuf
gzip Section 4.3.2	no change	.gz suffix
brotli Section 4.3.2	no change	.br suffix

Table 2

4.4. 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 NDJSON 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.

5. 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.

5.1. 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 3.4.7) 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 18.

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

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

Command: QLOGDIR=/srv/qlogs/ quicserverbinary

Should result in the quicserverbinary executable generating several logs files, o
ne for each QUIC connection.
Given two QUIC connections, with ODCID values "abcde" and "12345" respectively, t
his 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 n
amed 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 "grou
p_id" field with values "abcde" and "12345".
```

Figure 18: Environment variable examples for a QUIC implementation

5.2. 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 4.3.4). 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 7 before exposing qlogs to outside actors.

6. 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 4.1). 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 occurences 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.

7. 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/)

8. IANA Considerations

TODO: primarily the .well-known URI

9. References

9.1. Normative References

[QLOG-QUIC-HTTP3]
Marx, R., Ed., "QUIC and HTTP/3 event definitions for qlog", Work in Progress, Internet-Draft, draft-marx-qlog-event-definitions-quic-h3-02, 2 November 2020, <https://tools.ietf.org/html/draft-marx-qlog-event-definitions-quic-h3-02>.

9.2. Informative References

[RFC1952] Deutsch, P., "GZIP file format specification version 4.3", RFC 1952, DOI 10.17487/RFC1952, May 1996, <https://www.rfc-editor.org/info/rfc1952>.

[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>.

[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/info/rfc4180>.

[rfc7049] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", RFC 7049, DOI 10.17487/RFC7049, October 2013, <https://www.rfc-editor.org/info/rfc7049>.

[RFC7932] Alakuijala, J. and Z. Szabadka, "Brotli Compressed Data Format", RFC 7932, DOI 10.17487/RFC7932, July 2016, <https://www.rfc-editor.org/info/rfc7932>.

[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/info/rfc8259>.

Appendix A. Change Log

A.1. 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.2. 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

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.

Author's Address

Robin Marx
Hasselt University

Email: robin.marx@uhasselt.be