

QUIC
Internet-Draft
Intended status: Standards Track
Expires: May 7, 2020

M. Duke
F5 Networks, Inc.
N. Banks
Microsoft
November 4, 2019

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

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 and proposes an optional protocol called QUIC-LB to exchange the parameters of that encoding. This framework also enables offload of other QUIC functions to trusted intermediaries, given the explicit cooperation of the QUIC server.

Status of This Memo

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

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

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

This Internet-Draft will expire on May 7, 2020.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents

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

Table of Contents

1. Introduction	3
1.1. Terminology	4
2. Protocol Objectives	5
2.1. Simplicity	5
2.2. Security	5
2.3. Robustness to Middleboxes	6
2.4. Load Balancer Chains	6
3. First CID octet	6
3.1. Config Rotation	6
3.2. Configuration Failover	7
3.3. Length Self-Description	7
4. Routing Algorithms	8
4.1. Plaintext CID Algorithm	9
4.1.1. Load Balancer Actions	9
4.1.2. Server Actions	9
4.2. Obfuscated CID Algorithm	10
4.2.1. Load Balancer Actions	10
4.2.2. Server Actions	11
4.3. Stream Cipher CID Algorithm	11
4.3.1. Load Balancer Actions	12
4.3.2. Server Actions	12
4.4. Block Cipher CID Algorithm	13
4.4.1. Load Balancer Actions	13
4.4.2. Server Actions	14
5. Retry Service	14
5.1. Common Requirements	15
5.2. No-Shared-State Retry Service	15
5.2.1. Service Requirements	15
5.2.2. Server Requirements	17
5.3. Shared-State Retry Service	17
5.3.1. Service Requirements	19
5.3.2. Server Requirements	19
6. Configuration Requirements	19
7. Protocol Description	22
7.1. Out of band sharing	22
7.2. QUIC-LB Message Exchange	22
7.3. QUIC-LB Packet	22
7.4. Message Types and Formats	23

7.4.1.	ACK_LB Message	24
7.4.2.	FAIL Message	24
7.4.3.	ROUTING_INFO Message	24
7.4.4.	STREAM_CID Message	25
7.4.5.	BLOCK_CID Message	26
7.4.6.	SERVER_ID Message	27
7.4.7.	MODULUS Message	27
7.4.8.	PLAINTEXT Message	27
7.4.9.	RETRY_SERVICE_STATELESS message	28
7.4.10.	RETRY_SERVICE_STATEFUL message	28
8.	Security Considerations	28
8.1.	Outside attackers	29
8.2.	Inside Attackers	29
9.	IANA Considerations	29
10.	References	29
10.1.	Normative References	30
10.2.	Informative References	30
Appendix A.	Acknowledgments	30
Appendix B.	Change Log	30
B.1.	Since draft-duke-quic-load-balancers-05	30
B.2.	Since draft-duke-quic-load-balancers-04	30
B.3.	Since draft-duke-quic-load-balancers-03	30
B.4.	Since draft-duke-quic-load-balancers-02	31
B.5.	Since draft-duke-quic-load-balancers-01	31
B.6.	Since draft-duke-quic-load-balancers-00	31
Authors' Addresses	31

1. Introduction

QUIC packets 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 a method of coordination between QUIC servers and low-state load balancers to support connection IDs that encode routing information. It describes desirable properties of a solution, and then specifies a protocol that provides those properties. This protocol supports multiple encoding schemes that increase in complexity as they address paths between load balancer and server with weaker trust dynamics.

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.

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.

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.

When discussing security threats to QUIC-LB, we distinguish between "inside observers" and "outside observers." The former lie on the path between the load balancer and server, which often but not always lies inside the server's data center or cloud deployment. Outside observers are on the path between the load balancer and client. "Off-path" attackers, though not on any data path, may also be "inside" or "outside" depending on whether not they have network access to the server without intermediation by the load balancer and/or other security devices.

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 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.

However, concealing this mapping from inside observers is beyond the scope of QUIC-LB. By simply observing Link-Layer and/or Network-Layer addresses of packets containing distinct connection IDs, it is trivial to determine that they map to the same server, even if connection IDs are entirely random and do not encode routing information. Schemes that conceal these addresses (e.g., IPsec) can also conceal QUIC-LB messages.

Inside observers are generally able to mount Denial of Service (DoS) attacks on QUIC connections regardless of Connection ID schemes. However, QUIC-LB should protect against Denial of Service due to inside off-path attackers in cases where such attackers are possible.

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.

2.3. Robustness to Middleboxes

The path between load balancer and server may pass through middleboxes that could drop the coordination messages in this protocol. It is therefore advantageous to make messages resemble QUIC traffic as much as possible, as any viable path must obviously admit QUIC traffic.

2.4. Load Balancer Chains

While it is possible to construct a scheme that supports multiple low-state load balancers in the path, by using different parts of the connection ID to encode routing information for each load balancer, this use case is out of scope for QUIC-LB.

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 the configuration phase of that ID. QUIC-LB messages indicate the phase of the algorithm and parameters that they encode.

A new configuration may change one or more parameters of the old configuration, or change the algorithm used.

It is possible for servers to have mutually exclusive sets of supported algorithms, or for a transition from one algorithm to another to result in Fail Payloads. The four states encoded in these two bits allow two mutually exclusive server pools to coexist, and for each of them to transition to a new set of parameters.

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.

Servers MUST NOT generate new connection IDs using an old configuration when it has sent an Ack payload for a new configuration.

Load balancers SHOULD NOT use a codepoint to represent a new configuration until it takes precautions to make sure that all connections using IDs with an old configuration at that codepoint have closed or transitioned. They MAY drop connection IDs with the old configuration after a reasonable interval to accelerate this process.

3.2. Configuration Failover

If a server is configured to expect QUIC-LB messages, and it has not received these, it MUST generate connection IDs with the config rotation bits set to '11' and MUST use the "disable_migration" transport parameter in all new QUIC connections. It MUST 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, it is not negotiated between servers and load balancers. 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.

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":

- o The DCID might not be long enough for the routing algorithm to process.
- o The extracted server mapping might not correspond to an active server.
- o A field that should be all zeroes after decryption may not be so.

Load balancers MUST forward packets with long headers with 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. 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.

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

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 should not build 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 two participants: the load balancer and the server. The load balancer, in this description, generates configuration parameters. Note that in practice a third party configuration agent MAY assume this responsibility.

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.

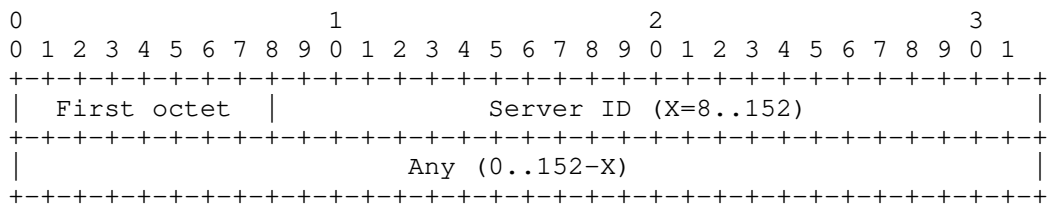


Figure 1: Plaintext CID Format

4.1.1. Load Balancer Actions

The load balancer selects a number of bytes of the server connection ID (SCID) that it will use to route to a given server, called the "routing bytes". The number of bytes MUST have enough entropy to have a different code point for each server.

The load balancer shares this value with servers, as explained in Section 7, along with the value that represents that server.

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

4.1.2. 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. Obfuscated CID Algorithm

The Obfuscated CID Algorithm makes an attempt to obscure the mapping of connections to servers to reduce linkability, while not requiring true encryption and decryption. The format is depicted in the figure below.

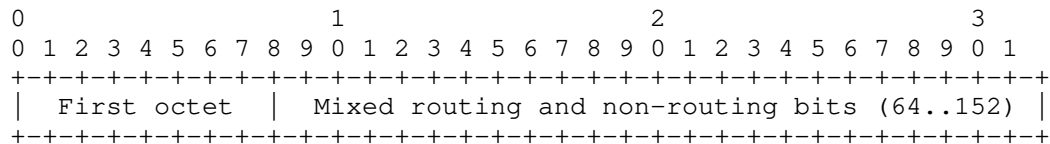


Figure 2: Obfuscated CID Format

4.2.1. Load Balancer Actions

The load balancer selects an arbitrary set of bits of the server connection ID (SCID) that it will use to route to a given server, called the "routing bits". The number of bits **MUST** have enough entropy to have a different code point for each server, and **SHOULD** have enough entropy so that there are many codepoints for each server.

The load balancer **MUST NOT** select a routing mask with more than 136 routing bits set to 1, which allows for the first octet and up to 2 octets for server purposes in a maximum-length connection ID.

The load balancer selects a divisor that **MUST** be larger than the number of servers. It **SHOULD** be large enough to accommodate reasonable increases in the number of servers. The divisor **MUST** be an odd integer so certain addition operations do not always produce an even number.

The load balancer also assigns each server a "modulus", an integer between 0 and the divisor minus 1. These **MUST** be unique for each server, and **SHOULD** be distributed across the entire number space between zero and the divisor.

The load balancer shares these three values with servers, as explained in Section 7.

Upon receipt of a QUIC packet, the load balancer extracts the selected bits of the SCID and expresses them as an unsigned integer of that length. The load balancer then divides the result by the chosen divisor. The modulus of this operation maps to the modulus for the destination server.

Note that any SCID that contains a server's modulus, plus an arbitrary integer multiple of the divisor, in the routing bits is routable to that server regardless of the contents of the non-routing bits. Outside observers that do not know the divisor or the routing bits will therefore have difficulty identifying that two SCIDs route to the same server.

Note also that not all Connection IDs are necessarily routable, as the computed modulus may not match one assigned to any server. These DCIDs are non-compliant as described above.

4.2.2. Server Actions

The server chooses a connection ID length. This MUST contain all of the routing bits and MUST be at least 9 octets to provide adequate entropy.

When a server needs a new connection ID, it adds an arbitrary nonnegative integer multiple of the divisor to its modulus, without exceeding the maximum integer value implied by the number of routing bits. The choice of multiple should appear random within these constraints.

The server encodes the result in the routing bits. It MAY put any other value into bits that used neither for routing nor config rotation. These bits SHOULD appear random to observers.

4.3. Stream Cipher CID Algorithm

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

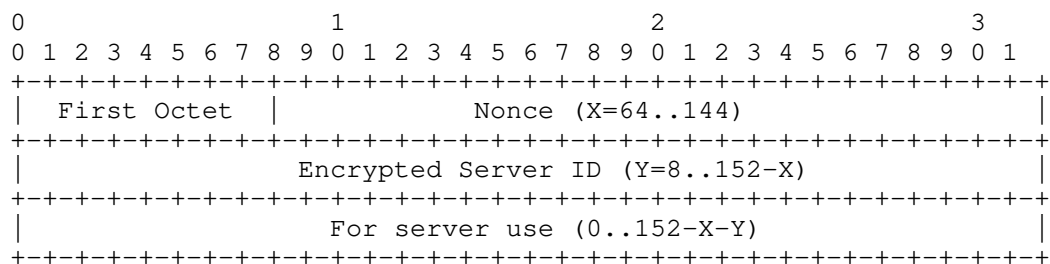


Figure 3: Stream Cipher CID Format

4.3.1. Load Balancer Actions

The load balancer 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 load balancer 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.

The load balancer shares these three values with servers, as explained in Section 7.

Upon receipt of a QUIC packet that is not of type Initial or 0-RTT, 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 server ID using 128-bit AES Electronic Codebook (ECB) mode, much like QUIC header protection. 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 encrypted server id.

```
server_id = encrypted_server_id ^ 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 of the connection ID for the nonce, zero-pads it to 16 octets using the first 6 octets of the token, and uses this to decrypt the server ID in the twelfth and thirteenth octet.

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

4.3.2. 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 nonce and additional bits **MAY** encode additional information, but **SHOULD** appear essentially random to observers.

The server decrypts the server ID using 128-bit AES Electronic Codebook (ECB) mode, much like QUIC header protection. The nonce

octets are zero-padded to 16 octets using the as many of the first octets of the token as necessary. AES-ECB encrypts this nonce using its key to generate a mask which it applies to the server id.

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

4.4. 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.

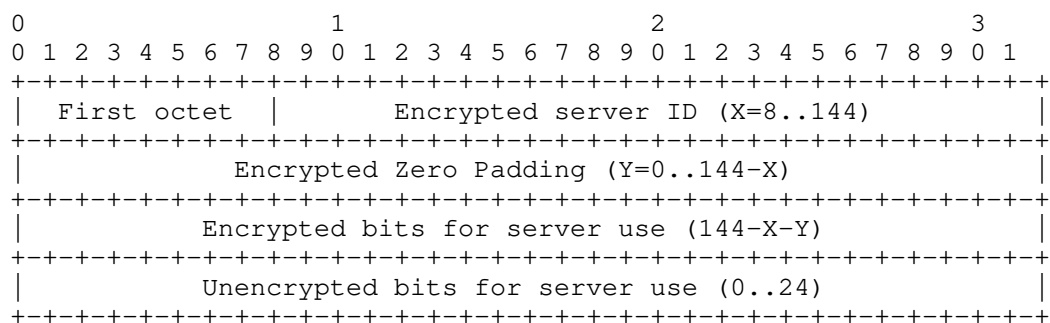


Figure 4: Block Cipher CID Format

4.4.1. Load Balancer Actions

The load balancer 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.

The load balancer selects a zero-padding length. This SHOULD be at least four octets to allow detection of non-compliant DCIDs. The server ID and zero-padding length MUST sum to no more than 16 octets. They SHOULD sum to no more than 12 octets, to provide servers adequate space to encode their own opaque data.

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

The load balancer shares these four values with servers, as explained in Section 7.

Upon receipt of a QUIC packet that is not of type Initial or 0-RTT, 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, zero padding, and opaque server data in that order. The load balancer uses the server ID octets for routing.

4.4.2. 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, zeroes into the zero-padding octets, and arbitrary bits into the remaining bits. These arbitrary bits MAY encode additional information. Bits in the first, 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. 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 the `original_connection_id` transport parameter, servers verify address ownership and clients verify that there is no "man in the middle" generating Retry packets.

As a trusted Retry Service is literally a "man in the middle," the service must communicate the `original_connection_id` 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 shared key, but has no such constraints.

Retry services MUST forward all non-Initial QUIC packets, as well as Initial packets from the server.

5.1. 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 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 unacceptably increase loads 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, or require different information.

5.2. 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. 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').

5.2.1. 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.

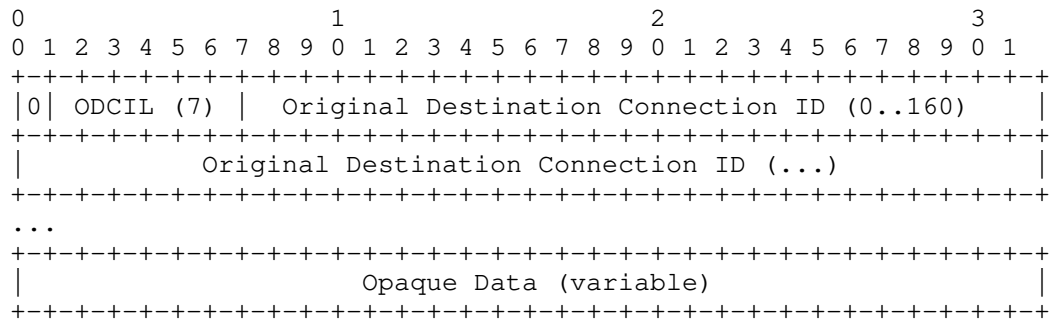


Figure 5: Format of non-shared-state retry service tokens

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.

Original Destination 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 encode a secure hash of the original destination connection ID field to verify that this field has not been edited.

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. It must also verify that the secure hash of the Connect ID is correct. If incorrect, the token is invalid.

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.

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 it has proof of address with its token.

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_connection_id` transport parameter, which preserves the normal QUIC signal to the client that there is an unauthorized man in the middle.

5.2.2. Server Requirements

A server behind a non-shared-state retry service MUST NOT send Retry packets.

Tokens sent in NEW_TOKEN frames MUST have the first bit be 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, assuming the format described in Section 5.2.1.

5.3. 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 not necessary.

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.

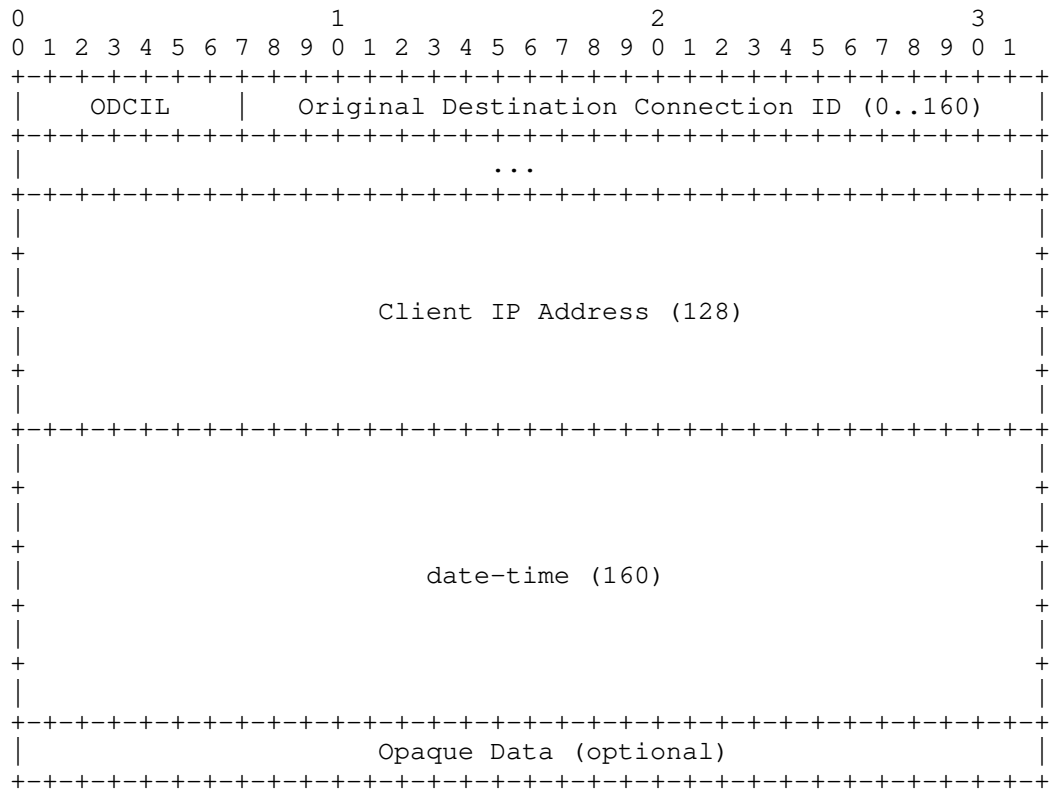


Figure 6: Cleartext format of shared-state retry tokens

The tokens have the following fields:

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

Original Destination Connection ID: This is copied from the field in the client Initial 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.

date-time: The date-time string is a total of 20 octets and encodes the time the token was generated. The format of date-time is described in Section 5.6 of [RFC3339]. This ASCII field MUST use the "Z" character for time-offset.

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

5.3.1. Service Requirements

The service MUST share a "token key" with all supported servers.

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 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.

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.

5.3.2. Server Requirements

The server MUST validate all tokens that arrive in Initial packets, as they may have bypassed the Retry service. It SHOULD use the date-time field to apply its expiration limits for tokens. This need not be 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.

A server MUST NOT send a Retry packet in response to an Initial packet that contains a retry token.

6. Configuration Requirements

QUIC-LB strives to minimize the configuration load to enable, as much as possible, a "plug-and-play" model. However, there are some configuration requirements based on algorithm and protocol choices above.

If there is any in-band communication, servers MUST be explicitly configured with the token of the load balancer they expect to interface with.

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

For Plaintext CID Routing, this consists of the Server ID and the routing bytes. The Server ID is unique to each server, and the routing bytes are global.

For Obfuscated CID Routing, this consists of the Routing Bits, Divisor, and Modulus. The Modulus is unique to each server, but the others MUST be global.

For Stream Cipher CID Routing, this consists of the Server ID, Server ID Length, Key, and Nonce Length. The Server ID is unique to each server, but the others MUST be global. The authentication token MUST be distributed out of band for this algorithm to operate.

For Block Cipher CID Routing, this consists of the Server ID, Server ID Length, Key, and Zero-Padding Length. The Server ID is unique to each server, but the others MUST be global.

A full QUIC-LB configuration MUST also specify the information content of the first CID octet and the presence and mode of any Retry Service.

The following pseudocode depicts 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;
enum      { in_band_config, out_of_band_config } config_method;
select (config_method) {
    case in_band_config: uint64 config_token;
    case out_of_band_config: null;
} config_method
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: null;
    case shared_state: uint8 key[16];
} retry_service_config;
enum      { none, plaintext, obfuscated, 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 obfuscated: struct {
        uint8 routing_bit_mask[19];
        uint16 divisor; /* Must be odd */
        uint16 modulus; /* 0..(divisor - 1) */
    } obfuscated_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 zero_padding_length; /* 0..(16 - server_id_length) */
        uint8 server_id[server_id_length];
        uint8 key[16];
    } block_cipher_config;
} routing_algorithm_config;
```

This specification allows for out-of-band dissemination of this configuration items, but also provides an in-band method for deployment models that need it.

7. Protocol Description

There are multiple means of configuration that correspond to differing deployment models and increasing levels of concern about the security of the load balancer-server path.

7.1. Out of band sharing

When there are concerns about the integrity of the path between load balancer and server, operators MAY share routing information using an out-of-band technique, which is out of the scope of this specification.

To simplify configuration, the global parameters can be shared out-of-band, while the load balancer sends the unique server IDs via the truncated message formats presented below.

7.2. QUIC-LB Message Exchange

QUIC-LB load balancers and servers exchange messages via the QUIC-LBv1 protocol, which uses the QUIC invariants with version number 0xF1000000. The QUIC-LB load balancers send the encoding parameters to servers and periodically retransmit until that server responds with an acknowledgement. Specifics of this retransmission are implementation-dependent.

7.3. QUIC-LB Packet

A QUIC-LB packet uses a long header. It carries configuration information from the load balancer and acknowledgements from the servers. They are sent when a load balancer boots up, detects a new server in the pool or needs to update the server configuration.

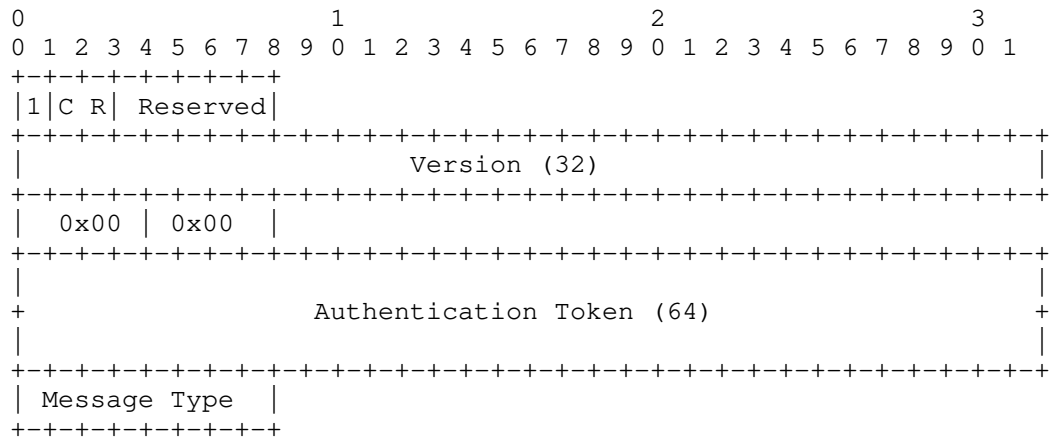


Figure 7: QUIC-LB Packet Format

The Version field allows QUIC-LB to use the Version Negotiation mechanism. All messages in this specification are specific to QUIC-LBv1. It should be set to 0xF1000000.

Load balancers MUST cease sending QUIC-LB packets of this version to a server when that server sends a Version Negotiation packet that does not advertise the version.

The length of the DCIL and SCIL fields are 0x00.

CR The 2-bit CR field indicates the Config Rotation described in Section 3.1.

Authentication Token The Authentication Token is an 8-byte field that both entities obtain at configuration time. It is used to verify that the sender is not an inside off-path attacker. Servers and load balancers SHOULD silently discard QUIC-LB packets with an incorrect token.

Message Type The Message Type indicates the type of message payload that follows the QUIC-LB header.

7.4. Message Types and Formats

As described in Section 7.3, QUIC-LB packets contain a single message. This section describes the format and semantics of the QUIC-LB message types.

7.4.1. ACK_LB Message

A server uses the ACK_LB message (type=0x00) to acknowledge a QUIC-LB packet received from the load balancer. The ACK-LB message has no additional payload beyond the QUIC-LB packet header.

Load balancers SHOULD continue to retransmit a QUIC-LB packet until a valid ACK_LB message, FAIL message or Version Negotiation Packet is received from the server.

7.4.2. FAIL Message

A server uses the FAIL message (type=0x01) to indicate the configuration received from the load balancer is unsupported.

```

0                               1                               2                               3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|  Supp. Type  |  Supp. Type  |  ...  |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

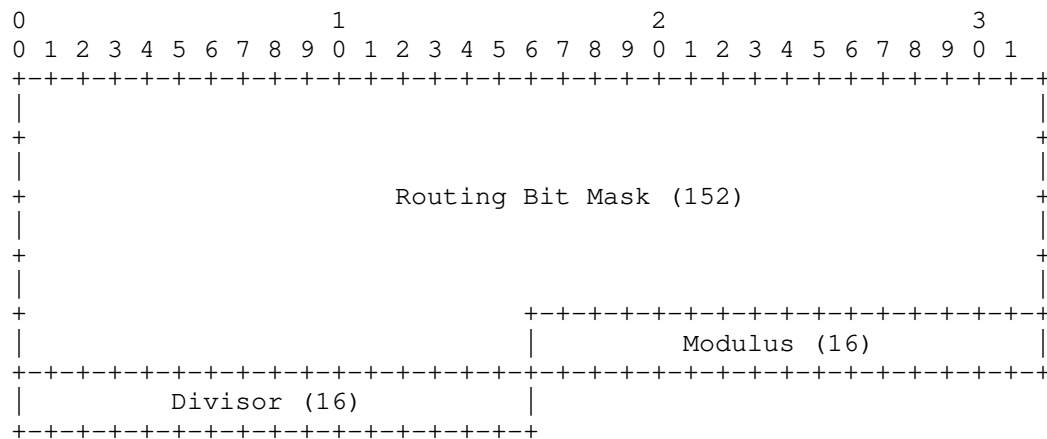
Servers MUST send a FAIL message upon receipt of a message type which they do not support, or if they do not possess all of the implied out-of-band configuration to support a particular message type.

The payload of the FAIL message consists of a list of all the message types supported by the server.

Upon receipt of a FAIL message, Load Balancers MUST either send a QUIC-LB message the server supports or remove the server from the server pool.

7.4.3. ROUTING_INFO Message

A load balancer uses the ROUTING_INFO message (type=0x02) to exchange all the parameters for the Obfuscated CID algorithm.



Routing Bit Mask The Routing Bit Mask encodes a '1' at every bit position in the server connection ID that will encode routing information.

These bits, along with the Modulus and Divisor, are chosen by the load balancer as described in Section 4.2.

7.4.4. STREAM_CID Message

A load balancer uses the `STREAM_CID` message (type=0x03) to exchange all the parameters for using Stream Cipher CIDs.

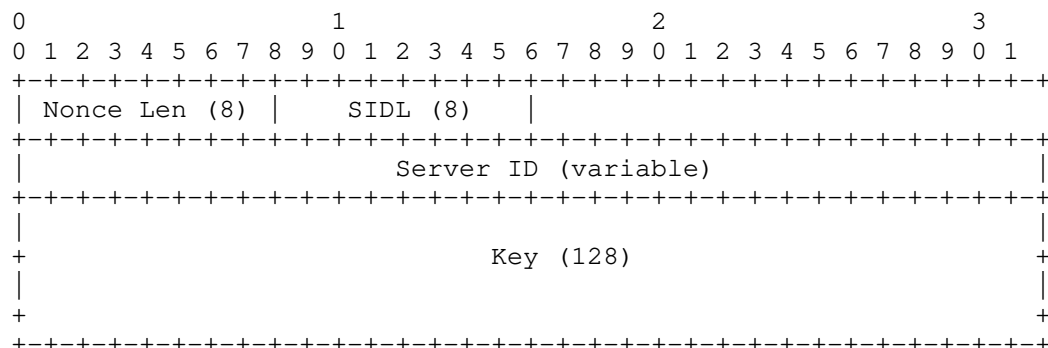


Figure 8: Stream CID Payload

Nonce Len The Nonce Len field is a one-octet unsigned integer that describes the nonce length necessary to use this routing algorithm, in octets.

SIDL The SIDL field is a one-octet unsigned integer that describes the server ID length necessary to use this routing algorithm, in octets.

Server ID The Server ID is the unique value assigned to the receiving server. Its length is determined by the SIDL field.

Key The Key is an 16-octet field that contains the key that the load balancer will use to decrypt server IDs on QUIC packets. See Section 8 to understand why sending keys in plaintext may be a safe strategy.

7.4.5. BLOCK_CID Message

A load balancer uses the BLOCK_CID message (type=0x04) to exchange all the parameters for using Stream Cipher CIDs.

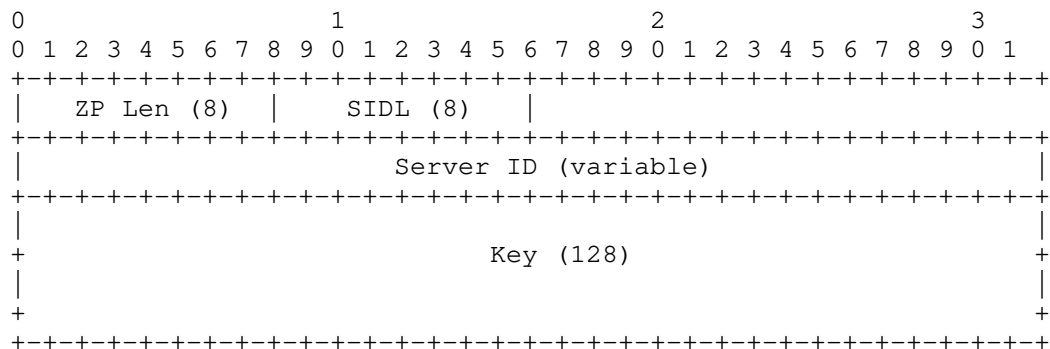


Figure 9: Block CID Payload

ZP Len The ZP Len field is a one-octet unsigned integer that describes the zero-padding length necessary to use this routing algorithm, in octets.

SIDL The SIDL field is a one-octet unsigned integer that describes the server ID length necessary to use this routing algorithm, in octets.

Server ID The Server ID is the unique value assigned to the receiving server. Its length is determined by the SIDL field.

Key The Key is an 16-octet field that contains the key that the load balancer will use to decrypt server IDs on QUIC packets. See Section 8 to understand why sending keys in plaintext may be a safe strategy.

7.4.6. SERVER_ID Message

A load balancer uses the SERVER_ID message (type=0x05) to exchange explicit server IDs.

```

0                               1                               2                               3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|   SIDL (8)   |   Server ID (variable)   |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

```

Load balancers send the SERVER_ID message when all global values for Stream or Block CIDs are sent out-of-band, so that only the server-unique values must be sent in-band. It also provides all necessary parameters for Plaintext CIDs. The fields are identical to their counterparts in the Section 7.4.4 payload.

7.4.7. MODULUS Message

A load balancer uses the MODULUS message (type=0x06) to exchange just the modulus used in the Obfuscated CID algorithm.

```

0                               1                               2                               3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|   Modulus (16)   |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

```

Load balancers send the MODULUS when all global values for Obfuscated CIDs are sent out-of-band, so that only the server-unique values must be sent in-band. The Modulus field is identical to its counterpart in the ROUTING_INFO message.

7.4.8. PLAINTEXT Message

A load balancer uses the PLAINTEXT message (type=0x07) to exchange all parameters needed for the Plaintext CID algorithm.

```

0                               1                               2                               3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|   SIDL (8)   |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|   Server ID (variable)   |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

```

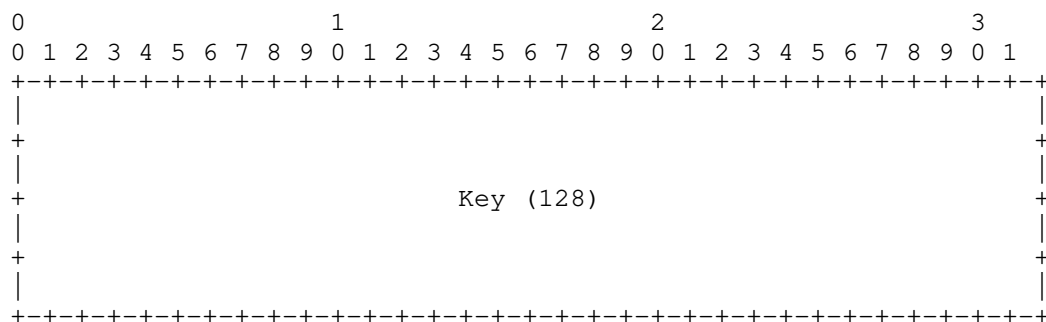
The SIDL field indicates the length of the server ID field. The Server ID field indicates the encoding that represents the destination server.

7.4.9. RETRY_SERVICE_STATELESS message

A no-shared-state retry service uses this message (type=0x08) to notify the server of the existence of this service. This message has no fields.

7.4.10. RETRY_SERVICE_STATEFUL message

A shared-state retry service uses this message (type=0x09) to tell the server about its existence, and share the key needed to decrypt server-generated retry tokens.



8. Security Considerations

QUIC-LB is intended to preserve routability and prevent linkability. Attacks on the protocol would compromise at least one of these objectives.

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. 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. Doing so might falsely suggest to the client that said CIDs were generated in a secure fashion.

A routability attack would inject QUIC-LB messages so that load balancers incorrectly route QUIC connections.

A linkability attack would find some means of determining that two connection IDs route to the same server. As described above, there

is no scheme that strictly prevents linkability for all traffic patterns, and therefore efforts to frustrate any analysis of server ID encoding have diminishing returns.

8.1. Outside attackers

For an outside attacker to break routability, it must inject packets that correctly guess the 64-bit token, and servers must be reachable from these outside hosts. Load balancers SHOULD drop QUIC-LB packets that arrive on its external interface.

Off-path outside attackers cannot observe connection IDs, and will therefore struggle to link them.

On-path outside attackers might try to link connection IDs to the same QUIC connection. The Encrypted CID algorithm provides robust entropy to making any sort of linkage. The Obfuscated CID obscures the mapping and prevents trivial brute-force attacks to determine the routing parameters, but does not provide robust protection against sophisticated attacks.

8.2. Inside Attackers

As described above, on-path inside attackers 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.

On-path inside attackers can break routability for new and migrating connections by copying the token from QUIC-LB messages. From this privileged position, however, there are many other attacks that can break QUIC connections to the server during the handshake.

Off-path inside attackers cannot observe connection IDs to link them. To successfully break routability, they must correctly guess the token.

9. IANA Considerations

There are no IANA requirements.

10. References

10.1. Normative References

[QUIC-TRANSPORT]

Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport (work in progress).

[RFC3339] Klyne, G. and C. Newman, "Date and Time on the Internet: Timestamps", RFC 3339, DOI 10.17487/RFC3339, July 2002, <<https://www.rfc-editor.org/info/rfc3339>>.

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. Acknowledgments

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-load-balancers-05

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

B.2. Since draft-duke-quic-load-balancers-04

- o Added standard for retry services

B.3. Since draft-duke-quic-load-balancers-03

- o Renamed Plaintext CID algorithm as Obfuscated CID
- o Added new Plaintext CID algorithm

- o Updated to allow 20B CIDs
 - o Added self-encoding of CID length
- B.4. Since draft-duke-quic-load-balancers-02
- o Added Config Rotation
 - o Added failover mode
 - o Tweaks to existing CID algorithms
 - o Added Block Cipher CID algorithm
 - o Reformatted QUIC-LB packets
- B.5. Since draft-duke-quic-load-balancers-01
- o Complete rewrite
 - o Supports multiple security levels
 - o Lightweight messages
- B.6. Since draft-duke-quic-load-balancers-00
- o Converted to markdown
 - o 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: Standards Track
Expires: May 7, 2020

T. Pauly
E. Kinnear
Apple Inc.
D. Schinazi
Google LLC
November 04, 2019

An Unreliable Datagram Extension to QUIC
draft-pauly-quic-datagram-05

Abstract

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

Discussion of this work is encouraged to happen on the QUIC IETF mailing list quic@ietf.org [1] or on the GitHub repository which contains the draft: <https://github.com/tfpaully/draft-pauly-quic-datagram> [2].

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 May 7, 2020.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

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

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

Table of Contents

1. Introduction	2
1.1. Specification of Requirements	3
2. Motivation	3
3. Transport Parameter	4
4. Datagram Frame Type	5
5. Behavior and Usage	5
5.1. Acknowledgement Handling	6
5.2. Flow Control	6
5.3. Congestion Control	6
6. Security Considerations	7
7. IANA Considerations	7
8. Acknowledgments	7
9. References	8
9.1. Normative References	8
9.2. Informative References	8
9.3. URIs	8
Authors' Addresses	9

1. Introduction

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

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

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

Discussion of this work is encouraged to happen on the QUIC IETF mailing list quic@ietf.org [3] or on the GitHub repository which

contains the draft: <https://github.com/tfpauly/draft-pauly-quic-datagram> [4].

1.1. Specification of Requirements

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

2. Motivation

Transmitting unreliable data over QUIC provides benefits over existing solutions:

- o Applications that open both a reliable TLS stream and an unreliable DTLS flow to the same peer can benefit by sharing a single handshake and authentication context between a reliable QUIC stream and flow of unreliable QUIC datagrams. This can reduce the latency required for handshakes.
- o QUIC uses a more nuanced loss recovery mechanism than the DTLS handshake, which has a basic packet loss retransmission timer. This may allow loss recovery to occur more quickly for QUIC data.
- o QUIC datagrams, while unreliable, can support acknowledgements, allowing applications to be aware of whether a datagram was successfully received.
- o QUIC datagrams are subject to QUIC congestion control, allowing applications to avoid implementing their own.

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

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

3. Transport Parameter

Support for receiving the DATAGRAM frame types is advertised by means of a QUIC Transport Parameter (name=max_datagram_frame_size, value=0x0020). The max_datagram_frame_size transport parameter is an integer value (represented as a variable-length integer) that represents the maximum size of a DATAGRAM frame (including the frame type, length, and payload) the endpoint is willing to receive, in bytes. An endpoint that includes this parameter supports the DATAGRAM frame types and is willing to receive such frames on this connection. Endpoints MUST NOT send DATAGRAM frames until they have sent and received the max_datagram_frame_size transport parameter. Endpoints MUST NOT send DATAGRAM frames of size strictly larger than the value of max_datagram_frame_size the endpoint has received from its peer. An endpoint that receives a DATAGRAM frame when it has not sent the max_datagram_frame_size transport parameter MUST terminate the connection with error `PROTOCOL_VIOLATION`. An endpoint that receives a DATAGRAM frame that is strictly larger than the value it sent in its max_datagram_frame_size transport parameter MUST terminate the connection with error `PROTOCOL_VIOLATION`. Endpoints that wish to use DATAGRAM frames need to ensure they send a max_datagram_frame_size value sufficient to allow their peer to use them. It is RECOMMENDED to send the value 65536 in the max_datagram_frame_size transport parameter as that indicates to the peer that this endpoint will accept any DATAGRAM frame that fits inside a QUIC packet.

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

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

4. Datagram Frame Type

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

The DATAGRAM frame is structured as follows:

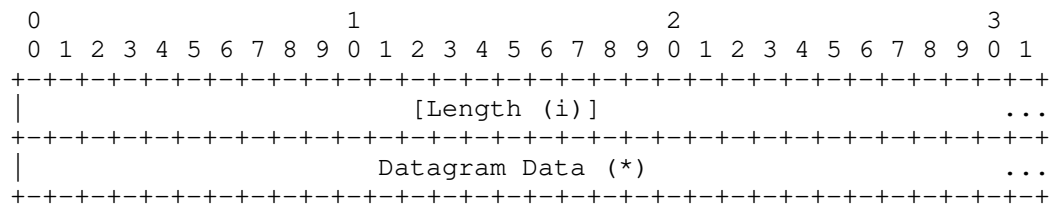


Figure 1: DATAGRAM Frame Format

DATAGRAM frames contain the following fields:

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

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

5. Behavior and Usage

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

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

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

Application protocols using datagrams are responsible for defining the semantics of the Datagram Data field, and how it is parsed. If the application protocol supports the coexistence of multiple

entities using datagrams inside a single QUIC connection, it may need a mechanism to allow demultiplexing between them. For example, using datagrams with HTTP/3 involves prepending a flow identifier to all datagrams, see [I-D.schinazi-quic-h3-datagram].

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

5.1. Acknowledgement Handling

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

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

5.2. Flow Control

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

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

5.3. Congestion Control

DATAGRAM frames employ the QUIC connection's congestion controller. As a result, a connection may be unable to send a DATAGRAM frame generated by the application until the congestion controller allows

it [I-D.ietf-quic-recovery]. The sender implementation **MUST** either delay sending the frame until the controller allows it or drop the frame without sending it (at which point it **MAY** notify the application).

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

6. Security Considerations

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

7. IANA Considerations

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

Value: 0x0020 (if this document is approved)

Parameter Name: max_datagram_frame_size

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

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

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

Frame Name: DATAGRAM

Specification: Unreliable application data

8. Acknowledgments

Thanks to Ian Swett, who inspired this proposal.

9. References

9.1. Normative References

- [I-D.ietf-quic-recovery]
Iyengar, J. and I. Swett, "QUIC Loss Detection and Congestion Control", draft-ietf-quic-recovery-23 (work in progress), September 2019.
- [I-D.ietf-quic-transport]
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-23 (work in progress), September 2019.

9.2. Informative References

- [I-D.schinazi-quic-h3-datagram]
Schinazi, D., "Using QUIC Datagrams with HTTP/3", draft-schinazi-quic-h3-datagram-01 (work in progress), October 2019.
- [RFC0768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, DOI 10.17487/RFC0768, August 1980, <<https://www.rfc-editor.org/info/rfc768>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <<https://www.rfc-editor.org/info/rfc6347>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.

9.3. URIs

- [1] <mailto:quic@ietf.org>
- [2] <https://github.com/tfpaully/draft-paully-quic-datagram>
- [3] <mailto:quic@ietf.org>
- [4] <https://github.com/tfpaully/draft-paully-quic-datagram>

Authors' Addresses

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

Email: tpauly@apple.com

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

Email: ekinnear@apple.com

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

Email: dschinazi.ietf@gmail.com

QUIC Working Group
Internet-Draft
Intended status: Informational
Expires: May 7, 2020

D. Schinazi
Google LLC
E. Rescorla
Mozilla
November 04, 2019

Compatible Version Negotiation for QUIC
draft-schinazi-quic-version-negotiation-02

Abstract

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

Discussion of this work is encouraged to happen on the QUIC IETF mailing list quic@ietf.org [1] or on the GitHub repository which contains the draft: <http://github.com/ekr/draft-schinazi-quic-version-negotiation> [2].

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 May 7, 2020.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction	2
2. Conventions and Definitions	3
3. Version Negotiation Mechanism	3
4. Version Negotiation Transport Parameter	4
5. Version Downgrade Prevention	6
6. Supported Versions	7
7. Compatible Versions	7
8. Security Considerations	7
9. IANA Considerations	7
10. References	8
10.1. Normative References	8
10.2. URIs	8
Authors' Addresses	8

1. Introduction

QUIC [QUIC] does not provide a complete version negotiation (VN) mechanism; the VN packet only allows the server to indicate that the version the client offered is unacceptable, but doesn't allow the client to safely make use of that information to create a new connection with a mutually supported version. With proper safety mechanisms in place, the VN packet can be part of a mechanism to allow two QUIC implementations to negotiate between two totally disjoint versions of QUIC, at the cost of an extra round trip. However, it is beneficial to avoid that cost whenever possible, especially given that most incremental versions are broadly similar to the the previous version.

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

Discussion of this work is encouraged to happen on the QUIC IETF mailing list quic@ietf.org [3] or on the GitHub repository which

contains the draft: <http://github.com/ekr/draft-schinazi-quic-version-negotiation> [4].

2. Conventions and Definitions

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

3. Version Negotiation Mechanism

The mechanism defined in this document is straightforward: the client maintains a list of QUIC versions it supports, ordered by preference. Its Initial packet is sent using the version that the server is most likely to support (in the absence of other information, this will often be the oldest version the client supports); that Initial packet then lists all compatible versions (Section 7) that the client supports in the Compatible Version fields of its transport parameters (Figure 1). Note that the client's compatible version list always contains its currently attempted version.

- o If the server supports one of the client's compatible versions, it selects a version it supports from the client's compatible version list. It then responds with that version in all of its future packets (except for Retry, as below).
- o If the server does not support any of the client's compatible versions, it sends a Version Negotiation packet listing all the versions it supports.

If the server leverages compatible versions and responds with a different version from the client's currently attempted version, it MUST NOT select a version not offered by the client. The client MUST validate that the version in the server's packets is one of the compatible versions that it offered and that it matches the negotiated version in the server's transport parameters.

If the server sends a Retry, it MUST use the same version that the client provided in its Initial. Version negotiation takes place after the retry cycle is over.

In order for negotiation to complete successfully, the client's Initial packet (and initial CRYPTO frames) MUST be interpretable by the server. This implies that servers must retain the ability to process the Initial packet from older versions as long as they are

reasonably popular. This is not generally an issue in practice as long as the the overall structure of the protocol remains similar.

4. Version Negotiation Transport Parameter

This document registers a new transport parameter, "version_negotiation". The contents of this transport parameter depend on whether the client or server is sending it, and are shown below:

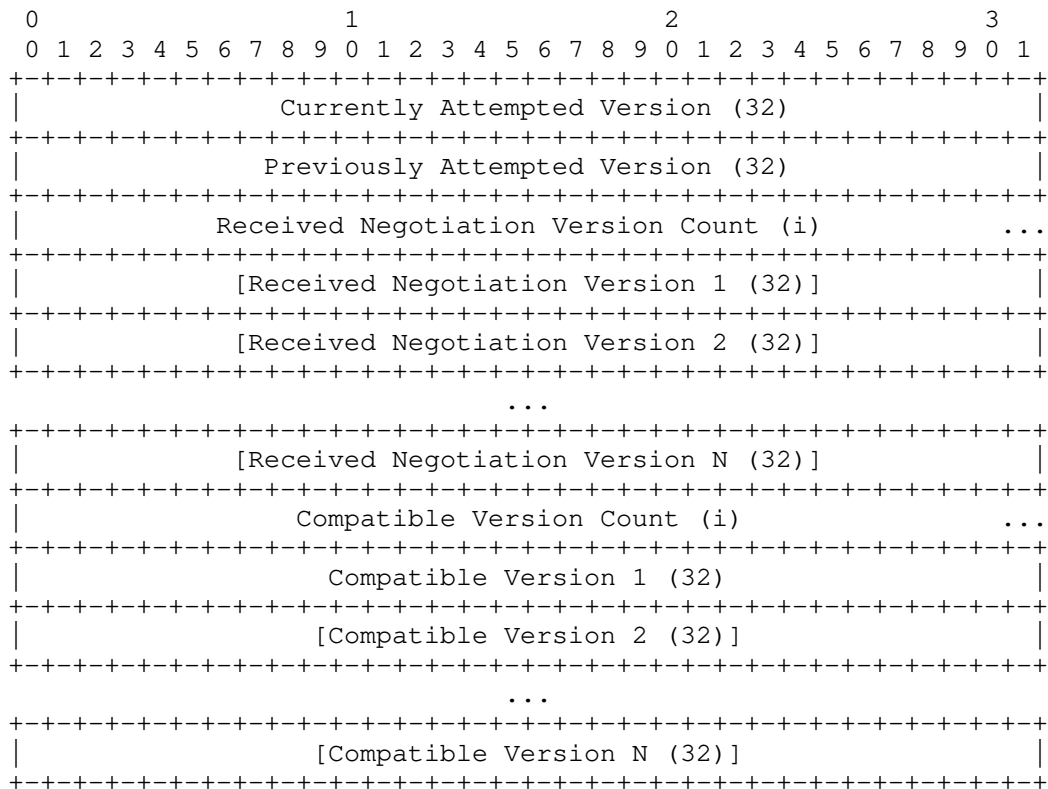


Figure 1: Client Transport Parameter Format

The content of each field is described below:

Currently Attempted Version: The version that the client is using in this Initial. This field **MUST** be equal to the value of the Version field in the long header that carries this transport parameter.

Previously Attempted Version: If the client is sending this Initial in response to a Version Negotiation packet, this field contains the version that the client used in the previous Initial packet that triggered the version negotiation packet. If the client did not receive a Version Negotiation packet, this field SHALL be all-zeroes.

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

Compatible Version Count: A variable-length integer specifying the number of Compatible Version fields following it. The client lists all versions compatible with Currently Attempted Version in the subsequent Compatible Version fields, ordered by descending preference. Note that the version in the Currently Attempted Version field MUST be included in the Compatible Version list to allow the client to communicate the currently attempted version's preference.

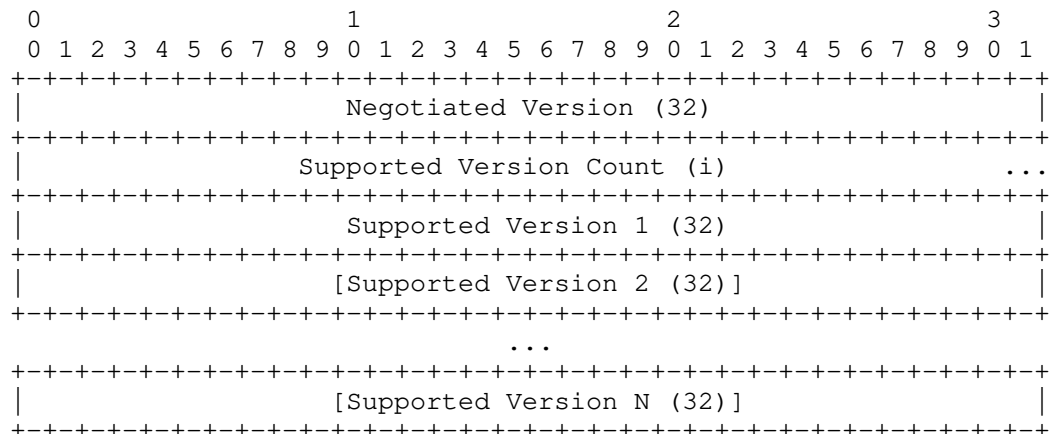


Figure 2: Server Transport Parameter Format

The content of each field is described below:

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

Supported Version Count: A variable-length integer specifying the number of Supported Version fields following it. The server encodes all versions it supports in the subsequent list, ordered by descending preference. Note that the version in the Negotiated Version field **MUST** be included in the Supported Version list.

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

5. Version Downgrade Prevention

Clients **MUST** ignore any received Version Negotiation packets that contain the version that they initially attempted.

Servers **MUST** validate that the client's "Currently Attempted Version" matches the version in the long header that carried the transport parameter. Similarly, clients **MUST** validate that the server's "Negotiated Version" matches the long header version. If an endpoint's validation fails, it **MUST** close the connection with an error of type `VERSION_NEGOTIATION_ERROR`.

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

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

6. Supported Versions

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

7. Compatible Versions

Two versions of QUIC A and B are said to be "compatible" if a version A Initial can be used to negotiate version B and vice versa. The most common scenario is a sequence of versions 1, 2, 3, etc. in which all the Initial packets have the same basic structure but might include specific extensions (especially inside the crypto handshake) that are only meaningful in some subset of versions and are ignored in others. Note that it is not possible to add new frame types in Initial packets because QUIC frames do not use a self-describing encoding, so unrecognized frame types cannot be parsed or ignored (see the Extension Frames section of [QUIC]).

When a new version of QUIC is defined, it is assumed to not be compatible with any other version unless otherwise specified. Implementations MUST NOT assume compatibility between version unless explicitly specified.

8. Security Considerations

The crypto handshake is already required to guarantee agreement on the supported parameters, so negotiation between compatible versions will have the security of the weakest common version.

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

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

9. IANA Considerations

If this document is approved, IANA shall assign the identifier 0x73DB for the "version_negotiation" transport parameter from the QUIC Transport Parameter Registry and the identifier 0x53F8 for

"VERSION_NEGOTIATION_ERROR" from the QUIC Transport Error Codes registry.

10. References

10.1. Normative References

- [QUIC] Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-23 (work in progress), September 2019.
- [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. URIs

- [1] <mailto:quic@ietf.org>
- [2] <http://github.com/ekr/draft-schinazi-quic-version-negotiation>
- [3] <mailto:quic@ietf.org>
- [4] <http://github.com/ekr/draft-schinazi-quic-version-negotiation>

Authors' Addresses

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

Email: dschinazi.ietf@gmail.com

Eric Rescorla
Mozilla

Email: ekr@rtfm.com