Workgroup: QUIC
Internet-Draft:
draft-ietf-quic-load-balancers-10
Published: 4 January 2022
Intended Status: Standards Track
Expires: 8 July 2022
Authors: M. Duke
F5 Networks, Inc.
N. Banks
Microsoft

QUIC-LB: Generating Routable QUIC Connection IDs

Abstract

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

Note to Readers

Discussion of this document takes place on the QUIC Working Group mailing list (quic@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/quic/.

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

Status of This Memo

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

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

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

This Internet-Draft will expire on 8 July 2022.

Copyright Notice

Table of Contents

1. Introduction

QUIC packets [RFC9000] usually contain a connection ID to allow endpoints to associate packets with different address/port 4-tuples to the same connection context. This feature makes connections robust in the event of NAT rebinding. QUIC endpoints usually designate the connection ID which peers use to address packets. Server-generated connection IDs create a potential need for out-of-band communication to support QUIC.

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

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

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

Aside from load balancing, a QUIC server may also desire to offload other protocol functions to trusted intermediaries. These

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

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

1.1. Terminology

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

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

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

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

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

1.2. Notation

All wire formats will be depicted using the notation defined in Section 1.3 of [RFC9000]. There is one addition: the function len() refers to the length of a field which can serve as a limit on a different field, so that the lengths of two fields can be concisely defined as limited to a sum, for example:

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

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

The example below illustrates the basic framework:

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

Figure 1: Example Format

2. Protocol Objectives

2.1. Simplicity

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

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

2.2. Security

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

QUIC-LB is relevant in the region between these extremes: when the information that two connection IDs map to the same server is helpful to linking two connection IDs. Obviously, any scheme that

transparently communicates this mapping to outside observers compromises QUIC's defenses against linkability.

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

3. First CID octet

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

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

3.1. Config Rotation

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

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

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

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

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

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

3.2. Configuration Failover

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

A load balancer that sees a connection ID with config rotation bits set to '11' MUST revert to 5-tuple routing. These connection IDs may be of any length; however, see Section 11.6 for limits on this length.

3.3. Length Self-Description

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

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

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

3.4. Format

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

Figure 2: First Octet Format

The first octet has the following fields:

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

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

4. Load Balancing Preliminaries

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

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

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

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

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

4.1. Unroutable Connection IDs

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

These client-generated CIDs might not conform to the expectations of the routing algorithm and therefore not be routable by the load balancer. Those that are not routable are "unroutable DCIDs" and receive similar treatment regardless of why they're unroutable:

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

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

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

All other DCIDs are routable.

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

Load balancers SHOULD drop short header packets with unroutable DCIDs.

When forwarding a packet with a long header and unroutable DCID, load balancers MUST use a fallback algorithm as specified in Section 4.2.

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

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

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

4.2. Fallback Algorithms

There are conditions described below where a load balancer routes a packet using a "fallback algorithm." It can choose any algorithm, without coordination with the servers, but the algorithm SHOULD be deterministic over short time scales so that related packets go to the same server. The design of this algorithm SHOULD consider the version-invariant properties of QUIC described in [RFC8999] to maximize its robustness to future versions of QUIC.

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

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

4.3. Server ID Allocation

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

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

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

The configuration agent SHOULD provide a means for servers to express the number of server IDs it can usefully employ, because a single routing address actually corresponds to multiple server entities (see Section 9.1).

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

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

4.4. CID format

All connection IDs use the following format:

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

Figure 3: CID Format

Each configuration specifies the length of the Server ID and Nonce fields, with limits defined for each algorithm. When using a given configuration, the server MUST generate CIDs of length equal to the lengths of these three fields.

The Server ID is assigned to each server in accordance with Section 4.3. Dynamically allocated SIDs are limited to seven octets or fewer. Statically allocated ones have different limits for each algorithm.

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 Nonce is selected by the server when it generates a CID. As the name implies, a server MUST use a nonce no more than once when generating a CID for a given server ID and unique set of configuration parameters.

The nonce length MUST be at least 4 octets. Additional limits on its length are different for each algorithm. See Section 11.6 for limits on nonce generation.

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

5. Routing Algorithms

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

5.1. Plaintext CID Algorithm

The Plaintext CID Algorithm makes no attempt to obscure the mapping of connections to servers, significantly increasing linkability.

5.1.1. Configuration Agent Actions

See Section 4.4.

5.1.2. Load Balancer Actions

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

5.1.3. Server Actions

When a server needs a new connection ID, it encodes one of its assigned server IDs in consecutive octets beginning with the second and chooses a nonce. This nonce MUST appear to be random (see Section 11.6).

5.2. Encrypted Short CID Algorithm

The Encrypted Short CID algorithm provides cryptographic protection at the cost of additional per-packet processing at the load balancer

to decrypt every incoming connection ID, unless the load balancer maintains state for the routing information of any given 4-tuple.

5.2.1. Configuration Agent Actions

The nonce length MUST be no fewer than 4 octets. The nonce SHOULD be at least as long as the server ID in order to save the load balancer an encryption pass; see below.

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

5.2.2. 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. See Section 11.6 for nonce generation considerations.

The server encrypts the server ID using the following four pass algorithm, which leverages 128-bit AES Electronic Codebook (ECB) mode, much like QUIC header protection.

In the text below, ^ is the XOR function and || is concatenation. The truncate() function takes the most significant octets of its argument, so that the XOR function operates on fields of the same length. The expand() function outputs 16 octets, with its first argument in the most significant bits, its second argument in the least significant bits, and zeros in all other positions. Thus,

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

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

1. The server concatenates the server ID and nonce to create plaintext_CID.

2. The server splits plaintext_CID into components left_0 and right_0 of equal length, splitting an odd octet in half if necessary. For example, 0x7040b81b55ccf3 would split into a left_0 of 0x7040b81 and right_0 of 0xb55ccf3.

3. Encrypt left_0. The encryption is 128-bit AES-ECB with the key provided by the configuration agent, and the plaintext argument is an expanded version of left_0 where left_0 constitutes the most significant bits, 0x01 is the least significant octet, and all other bits are zero.

4. XOR the most significant bits of the ciphertext with right_0 to form right_1.

   Thus steps 3 and 4 can be expressed as

   right_1 = right_0 ^ truncate(AES_ECB(key, expand(left_0, 0x01))

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

   left_1 = left_0 ^ truncate(AES_ECB(key, expand(right_1), 0x02))

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

   right_2 = right_1 ^ truncate(AES_ECB(key, expand(left_1, 0x03))

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

   left_2 = left_1 ^ truncate(AES_ECB(key, expand(right_2), 0x04))

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

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

```
server_id = 0x3144a
nonce = 0x9c69c275
key = 0xfdf726a9893ec05c0632d30z6680baf0

// step 1
plaintext_CID = 0x31441a9c69c275

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

// step 3
aes_input = 0x31441a900000000000000000000000001
ciphertext = 0xdea73834473e88afee51be7f6bdff0e7

// step 4
right_1 = 0xc69c275 ^ 0xdea7383 = 0x183b1f6

// step 5
aes_input = 0x183b1f600000000000000000000000002
aes_output = 0x15ab4a6f252c0283a0446c74c3f98860
left_1 = 0x31441a9 ^ 0x15ab4a6 = 0x24ef50f

// step 6
AES input = 0x24ef50f000000000000000000000003
AES output = 0xbeaca161e903ebb97cfda599a29ad8ff
right_2 = 0x183b1f6 ^ 0xbeaca16 = 0xa697be0

// step 7
AES input: = 0xa697be000000000000000000000004
AES output = 0x13ea04a5e3c707bf197e8fcbcd43ef98
left_2 = 0x24ef50f ^ 0x13ea04a = 0x3705545

// step 8
cid = first_octet || left_2 || right_2 = 0x073705545a697be0
```

5.2.3. 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 server ID. The nonce immediately follows.

The load balancer decrypts the nonce and the server ID using the reverse of the algorithm above.

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

```
left_1 = left_2 ^ truncate(AES_ECB(key, expand(right_2), 0x04))
right_1 = right_2 ^ truncate(AES_ECB(key, expand(left_1, 0x03))
left_0 = left_1 ^ truncate(AES_ECB(key, expand(right_1), 0x02))
```

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

```
right_0 = right_1 ^ truncate(AES_ECB(key, expand(left_0, 0x01)))
```

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

5.3. Encrypted Long CID Algorithm

The Encrypted Long CID Algorithm, by using a full 16 octets of plaintext and a 128-bit cipher, protects the server ID with a single encryption pass. However, it also requires connection IDs of at least 17 octets, increasing overhead of client-to-server packets.

5.3.1. Configuration Agent Actions

The server ID length MUST be no more than 12 octets. The nonce and server ID MUST sum to at least 16 octets.

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

5.3.2. Load Balancer Actions

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

The first octets of the plaintext contains the server id.

5.3.3. Server Actions

The server encrypts both its server ID and enough octets in a nonce to form a 16-octet block using the configured AES-ECB key. Note that any remaining octets in the nonce are transmitted as plaintext, and should consider the constraints in Section 11.6.

6. ICMP Processing

For protocols where 4-tuple load balancing is sufficient, it is straightforward to deliver ICMP packets from the network to the

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

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

Alternatively, servers MAY implement the technique described in Section 14.4.1 of [RFC9000] to increase the likelihood a Source Connection ID is included in ICMP responses to Path Maximum Transmission Unit (PMTU) probes. Load balancers MAY parse the echoed packet to extract the Source Connection ID, if it contains a QUIC long header, and extract the Server ID as if it were in a Destination CID.

7. Retry Service

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

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

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

7.1. Common Requirements

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

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

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

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

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

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

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

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

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

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

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

7.1.1. Considerations for Non-Initial Packets

Initial Packets are especially effective at consuming server resources because they cause the server to create connection state. Even when mitigating this load with Retry Packets, the act of

validating an Initial Token and sending a Retry Packet is more expensive than the response to a non-Initial packet with an unknown Connection ID: simply dropping it and/or sending a Stateless Reset.

Nevertheless, a Retry Service in Active Mode might desire to shield servers from non-Initial packets that do not correspond to a previously admitted Initial Packet. This has a number of considerations.

- If a Retry Service maintains no per-flow state whatsoever, it cannot distinguish between valid and invalid non-Initial packets and MUST forward all non-Initial Packets to the server.

- For QUIC versions the Retry Service does not support and are present on the Allow-List (or absent from the Deny-List), the Retry Service cannot distinguish Initial Packets from other long headers and therefore MUST admit all long headers.

- If a Retry Service keeps per-flow state, it can identify 4-tuples that have been previously approved, admit non-Initial packets from those flows, and drop all others. However, dropping short headers will effectively break Address Migration and NAT Rebinding when in Active Mode, as post-migration packets will arrive with a previously unknown 4-tuple. This policy will also break connection attempts using any new QUIC versions that begin connections with a short header.

- If a Retry Service is integrated with a QUIC-LB routable load balancer, it can verify that the Destination Connection ID is routable, and only admit non-Initial packets with routable DCIDs. As the Connection ID encoding is invariant across QUIC versions, the Retry Service can do this for all short headers.

Nothing in this section prevents Retry Services from making basic syntax correctness checks on packets with QUIC versions that it understands (e.g., enforcing the Initial Packet datagram size minimum in version 1) and dropping packets that are not routable with the QUIC specification.

7.2. No-Shared-State Retry Service

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

7.2.1. Configuration Agent Actions

See Section 7.1.

7.2.2. Service Requirements

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

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

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

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

Figure 4: Format of non-shared-state retry service tokens

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

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

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

Opaque Data: This data contains the information necessary to authenticate the Retry token in accordance with the QUIC specification. A straightforward implementation would encode the Retry Source Connection ID, client IP address, and a timestamp in the Opaque Data. A more space-efficient implementation would use the Retry Source Connection ID and Client IP as associated data in an encryption operation, and encode only the timestamp and the authentication tag in the Opaque Data. If the Initial Packet has

altered the Connection ID or source IP address, authentication of the token will fail.

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

In active mode, the service MUST issue Retry packets for all Client initial packets that contain no token, or a token that has the first bit set to '1'. It MUST NOT forward the packet to the server. The service MUST validate all tokens with the first bit set to '0'. If successful, the service MUST forward the packet with the token intact. If unsuccessful, it MUST drop the packet. The Retry Service MAY send an Initial Packet containing a CONNECTION_CLOSE frame with the INVALID_TOKEN error code when dropping the packet.

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

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

7.2.3. Server Requirements

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

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

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

from the token cleartext for use in the transport parameter of the same name.

7.3. Shared-State Retry Service

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

Both server and service must have time synchronized with respect to one another to prevent tokens being incorrectly marked as expired, though tight synchronization is unnecessary.

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

- A 1 bit token type identifier.
- A 7 bit token key identifier.
- A 96 bit unique token number transmitted in clear text, but protected as part of the AEAD associated data.
- A token body, encoding the Original Destination Connection ID and the Timestamp, optionally followed by server specific Opaque Data.

The token protection uses an 128 bit representation of the source IP address from the triggering Initial packet. The client IP address is 16 octets. If an IPv4 address, the last 12 octets are zeroes. It also uses the Source Connection ID of the Retry packet, which will cause an authentication failure if it differs from the Destination Connection ID of the packet bearing the token.

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

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

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

The token body is encoded as follows:

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

Figure 5: Body of shared-state retry service tokens

The token body has the following fields:

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

ODCIL: The original destination connection ID length. Tokens in NEW_TOKEN frames do not have this field.

Original Destination Connection ID: The server or Retry Service copies this from the field in the client Initial packet. Tokens in NEW_TOKEN frames do not have this field.

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

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

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

Shared-state Retry Services use the AES-128-ECB cipher. Future standards could add new algorithms that use other ciphers to provide cryptographic agility in accordance with [RFC7696]. Retry Service and server implementations SHOULD be extensible to support new algorithms.

7.3.1. Token Protection with AEAD

On the wire, the token is presented as:

```
Shared-State Retry Service Token {
  Token Type (1),
  Key Sequence (7),
  Unique Token Number (96),
  Encrypted Shared-State Retry Service Token Body (64..),
  AEAD Integrity Check Value (128),
}
```

Figure 6: Wire image of shared-state retry service tokens

The tokens are protected using AES128-GCM as follows:

- The Key Sequence is the 7 bit identifier to retrieve the token key and IV.

- The AEAD IV, is a 96 bit data which produced by implementer's custom AEAD IV derivation function.

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

- The associated data is a formatted as a pseudo header by combining the cleartext part of the token with the IP address of the client. The format of the pseudoheader depends on whether the Token Type bit is '1' (a NEW_TOKEN token) or '0' (a Retry token).

```
Shared-State Retry Service Token Pseudoheader {
  IP Address (128),
  Token Type (1),
  Key Sequence (7),
  Unique Token Number (96),
  [RSCIL (8)],
  [Retry Source Connection ID (0..20)],
}
```

Figure 7: Psuedoheader for shared-state retry service tokens

RSCIL: The Retry Source Connection ID Length in octets. This field is only present when the Token Type is '0'.

Retry Source Connection ID: To create a Retry Token, populate this field with the Source Connection ID the Retry packet will use. To validate a Retry token, populate it with the Destination Connection ID of the Initial packet that carries the token. This field is only present when the Token Type is '0'.

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

*The AEAD Integrity Check Value(ICV), defined in Section 6 of [RFC4106], is computed as part of the AEAD encryption process, and is verified during decryption.

7.3.2. Configuration Agent Actions

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

7.3.3. Service Requirements

In inactive mode, the Retry service forwards all packets without further inspection or processing. The rest of this section only applies to a service in active mode.

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

The service MUST generate Retry tokens with the format described above when it receives a client Initial packet with no token.

If there is a token of either type, the service MUST attempt to decrypt it.

To decrypt a packet, the service checks the Token Type and constructs a pseudoheader with the appropriate format for that type, using the bearing packet's Destination Connection ID to populate the Retry Source Connection ID field, if any.

A token is invalid if:

*it uses unknown key sequence,

*the AEAD ICV does not match the expected value (By construction, it will only match if the client IP Address, and any Retry Source Connection ID, also matches),

*the ODCIL, if present, is invalid for a client-generated CID (less than 8 or more than 20 in QUIC version 1),

* the Timestamp of a token points to time in the past (however, in order to allow for clock skew, it SHOULD NOT consider tokens to be expired if the Timestamp encodes a few seconds in the past), or

* the port number, if present, does not match the source port in the encapsulating UDP header.

Packets with valid tokens MUST be forwarded to the server.

The service MUST drop packets with invalid tokens. If the token is of type '1' (NEW_TOKEN), it MUST respond with a Retry packet. If of type '0', it MUST NOT respond with a Retry packet.

7.3.4. Server Requirements

The server MAY issue Retry or NEW_TOKEN tokens in accordance with [RFC9000]. When doing so, it MUST follow the format above.

The server MUST validate all tokens that arrive in Initial packets, as they may have bypassed the Retry service. It determines validity using the procedure in Section 7.3.3.

If a valid Retry token, the server populates the original_destination_connection_id transport parameter using the corresponding token field. It populates the retry_source_connection_id transport parameter with the Destination Connection ID of the packet bearing the token.

In all other respects, the server processes both valid and invalid tokens in accordance with [RFC9000].

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

8. Configuration Requirements

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

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

All algorithm configurations can have a server ID length, nonce length, and key. However, for Plaintext CID, there is no key.

The load balancer MUST receive the full table of mappings, and each server must receive its assigned SID(s), from the configuration agent.

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

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

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

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

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

9. Additional Use Cases

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

9.1. Load balancer chains

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

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

9.2. Moving connections between servers

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

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

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

10. Version Invariance of QUIC-LB

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

The server ID encodings, and requirements for their handling, are designed to be QUIC version independent (see [RFC8999]). A QUIC-LB load balancer will generally not require changes as servers deploy new versions of QUIC. However, there are several unlikely future design decisions that could impact the operation of QUIC-LB.

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

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

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

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

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

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

While this document does not update the commitments in [RFC8999], the additional assumptions are minimal and narrowly scoped, and provide a likely set of constants that load balancers can use with minimal risk of version- dependence.

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

Some load balancers might inspect elements of the Server Name Indication (SNI) extension in the TLS Client Hello to make a routing

decision. Note that the format and cryptographic protection of this information may change in future versions or extensions of TLS or QUIC, and therefore this functionality is inherently not version-invariant.

11. Security Considerations

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

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

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

11.1. Attackers not between the load balancer and server

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

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

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

11.2. Attackers between the load balancer and server

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

11.3. Multiple Configuration IDs

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

11.4. Limited configuration scope

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

To avoid this, the configuration agent SHOULD issue QUIC-LB configurations to mutually distrustful servers that have different keys for encryption algorithms. In many cases, the load balancers can distinguish these configurations by external IP address.

However, assigning multiple entities to an IP address is complimentary with concealing DNS requests (e.g., DoH [RFC8484]) and the TLS Server Name Indicator (SNI) ([I-D.ietf-tls-esni]) to obscure the ultimate destination of traffic. While the load balancer's fallback algorithm (Section 4.2) can use the SNI to make a routing decision on the first packet, there are three ways to route subsequent packets:

* all co-tenants can use the same QUIC-LB configuration, leaking the server mapping to each other as described above;

* co-tenants can be issued one of up to three configurations distinguished by the config rotation bits (Section 3.1), exposing information about the target domain to the entire network; or

* tenants can use 4-tuple routing in their CIDs (in which case they SHOULD disable migration in their connections), which neutralizes the value of QUIC-LB but preserves privacy.

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

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

11.5. Stateless Reset Oracle

Section 21.9 of [RFC9000] discusses the Stateless Reset Oracle attack. For a server deployment to be vulnerable, an attacking client must be able to cause two packets with the same Destination CID to arrive at two different servers that share the same cryptographic context for Stateless Reset tokens. As QUIC-LB requires deterministic routing of DCIDs over the life of a connection, it is a sufficient means of avoiding an Oracle without additional measures.

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

11.6. Connection ID Entropy

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

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

If servers simply increment the nonce by one with each generated connection ID, then it is safe to use the existing keys until any server's nonce counter exhausts the allocated space and rolls over. To maximize entropy, servers SHOULD start with a random nonce value, in which case the configuration is usable until the nonce value wraps around to zero and then reaches the initial value again.

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

Both the Plaintext CID and Encrypted Long CID algorithms send parts of their nonce in plaintext. Servers MUST generate nonces so that

the plaintext portion appears to be random. Observable correlations between plaintext nonces would provide trivial linkability between individual connections, rather than just to a common server.

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

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

11.7. Shared-State Retry Keys

The Shared-State Retry Service defined in Section 7.3 describes the format of retry tokens or new tokens protected and encrypted using AES128-GCM. Each token includes a 96 bit randomly generated unique token number, and an 8 bit identifier used to get the AES-GCM encryption context. The AES-GCM encryption context contains a 128 bit key and an AEAD IV. There are three important security considerations for these tokens:

- An attacker that obtains a copy of the encryption key will be able to decrypt and forge tokens.
- Attackers may be able to retrieve the key if they capture a sufficently large number of retry tokens encrypted with a given key.
- Confidentiality of the token data will fail if separate tokens reuse the same 96 bit unique token number and the same key.

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

Section 6.6 of [RFC9001] states that "Endpoints MUST count the number of encrypted packets for each set of keys. If the total number of encrypted packets with the same key exceeds the confidentiality limit for the selected AEAD, the endpoint MUST stop using those keys." It goes on with the specific limit: "For AEAD_AES_128_GCM and AEAD_AES_256_GCM, the confidentiality limit is 2^23 encrypted packets; see Appendix B.1." It is prudent to adopt

the same limit here, and configure the service in such a way that no more than 2^23 tokens are generated with the same key.

In order to protect against collisions, the 96 bit unique token numbers should be generated using a cryptographically secure pseudorandom number generator (CSPRNG), as specified in Appendix C.1 of the TLS 1.3 specification [RFC8446]. With proper random numbers, if fewer than 2^40 tokens are generated with a single key, the risk of collisions is lower than 0.001%.

12. IANA Considerations

There are no IANA requirements.

13. References

13.1. Normative References

[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>.

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

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

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

13.2. Informative References

[I-D.draft-ietf-tls-dtls13] Rescorla, E., Tschofenig, H., and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3", Work in Progress, Internet-Draft, draft-ietf-tls-dtls13-43, 30 April 2021, <https://www.ietf.org/archive/id/draft-ietf-tls-dtls13-43.txt>.

[I-D.ietf-tls-dtls-connection-id] Rescorla, E., Tschofenig, H., Fossati, T., and A. Kraus, "Connection Identifiers for DTLS 1.2", Work in Progress, Internet-Draft, draft-ietf-tls-dtls-connection-id-13, 22 June 2021, <https://www.ietf.org/archive/id/draft-ietf-tls-dtls-connection-id-13.txt>.

[I-D.ietf-tls-esni] Rescorla, E., Oku, K., Sullivan, N., and C. A. Wood, "TLS Encrypted Client Hello", Work in Progress, Internet-Draft, draft-ietf-tls-esni-13, 12 August 2021, <https://www.ietf.org/archive/id/draft-ietf-tls-esni-13.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>.

[RFC4106] Viega, J. and D. McGrew, "The Use of Galois/Counter Mode (GCM) in IPsec Encapsulating Security Payload (ESP)", RFC 4106, DOI 10.17487/RFC4106, June 2005, <https://www.rfc-editor.org/info/rfc4106>.

[RFC4347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security", RFC 4347, DOI 10.17487/RFC4347, April 2006, <https://www.rfc-editor.org/info/rfc4347>.

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

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

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

[RFC7983] Petit-Huguenin, M. and G. Salgueiro, "Multiplexing Scheme Updates for Secure Real-time Transport Protocol (SRTP) Extension for Datagram Transport Layer Security (DTLS)",

RFC 7983, DOI 10.17487/RFC7983, September 2016, <https://www.rfc-editor.org/info/rfc7983>.

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

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

[RFC9001] Thomson, M., Ed. and S. Turner, Ed., "Using TLS to Secure QUIC", RFC 9001, DOI 10.17487/RFC9001, May 2021, <https://www.rfc-editor.org/info/rfc9001>.

Appendix A. QUIC-LB YANG Model

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

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

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

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

  organization
    "IETF QUIC Working Group";

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

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

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

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

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

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

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

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

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

  description
    "QUIC-LB container.";

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

  typedef algorithm-type {
    type enumeration {
      enum plaintext {
        description "Plaintext CID Algorithm";
      }
      enum encrypted-short {
         description "Encrypted Short CID Algorithm";
      }
      enum encrypted-long {
        description "Encrypted Long CID Algorithm";
      }
    }
  }

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

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

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

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

leaf algorithm {
  type algorithm-type;
  mandatory true;
  description
    "The algorithm that encodes the server ID";
}

must 'cid-key or (algorithm = "plaintext")' {
  error-message "Encrypted algorithm requires key";
}

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

leaf server-id-length {
  type uint8 {
    range "1..15";
  }
  must '. <= (19 - ../nonce-length)' {
    error-message
      "Server ID and nonce lengths must sum to no more than 19.";
  }
  must '(../algorithm != "encrypted-long") or (. <= 12)' {
    error-message
      "encrypted-long requires server ID length <= 12.";
  }
  must '(../algorithm != "encrypted-long") or
```

```
            ((. + ../nonce-length) >= 16)' {
        error-message
          "For Encrypted Long CIDs, server ID length plus nonce length
           least 16";
      }
      mandatory true;
      description
        "Length (in octets) of a server ID. Further range-limited
         by sid-allocation, cid-key, and nonce-length.";
    }

    list server-id-mappings {
      key "server-id";
      description "Statically allocated Server IDs";

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

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

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

    leaf-list supported-versions {
      type uint32;
      description
        "QUIC versions that the retry service supports. If empty, ther
         is no retry service.";
    }

    leaf unsupported-version-default {
      type enumeration {
        enum allow {
          description "Unsupported versions admitted by default";
```

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

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

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

    leaf key-sequence-number {
      type uint8 {
        range "0..127";
      }
      mandatory true;
      description
        "Identifies the key used to encrypt the token";
    }

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

    leaf token-iv {
      type yang:hex-string {
        length 23;
      }
      mandatory true;
      description
        "8-byte IV to encrypt the token, encoded in 23 bytes";
    }
  }
}
```

```
  }
}
```

A.1. Tree Diagram

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

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

Appendix B. Load Balancer Test Vectors

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

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

This section uses the following abbreviations:

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

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

B.1. Plaintext Connection ID Algorithm

TBD

B.2. Encrypted Short Connection ID Algorithm

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

TBD

B.3. Encrypted Long Connection ID Algorithm

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

TBD

B.4. Shared State Retry Tokens

In this case, the shared-state retry token is issued by retry service, so the opaque data of shared-state retry token body would be null (Section 7.3).

```
LB configuration:
key_seq 0x00
encrypt_key 0x303132333435363738393031323334353
AEAD_IV 0x313233343536373839303132

Shared-State Retry Service Token Body:
ODCIL 0x12
RSCIL 0x10
port 0x1a0a
original_destination_connection_id 0x0c3817b544ca1c94313bba41757547eec93
retry_source_connection_id 0x0301e770d24b3b13070dd5c2a9264307
timestamp 0x0000000060c7bf4d

Shared-State Retry Service Token:
unique_token_number 0x59ef316b70575e793e1a8782
key_sequence 0x00
encrypted_shared_state_retry_service_token_body
0x7d38b274aa4427c7a1557c3fa666945931defc65da387a83855196a7cb73caac1e28e5
AEAD_ICV 0xf91174fdd711543a32d5e959867f9c22

AEAD related parameters:
client_ip_addr 127.0.0.1
client_port 6666
AEAD_nonce 0x68dd025f45616941072ab6b0
AEAD_associated_data 0x7f000001000000000000000000000059ef316b70575e793
```

Appendix C. Interoperability with DTLS over UDP

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

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

C.1. DTLS 1.0 and 1.2

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

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

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

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

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

C.2. DTLS 1.3

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

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

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

However, if the client does not include the connection_id extension in its ClientHello, the server is unable to use connection IDs. In this case, non- handshake packets will appear to contain random connection IDs and be routed randomly. Thus, unmodified QUIC-LB load balancers will not work with DTLS 1.3 if the client does not advertise support for connection IDs, or the server does not request the use of a compliant connection ID.

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

C.3. Future Versions of DTLS

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

Appendix D. Acknowledgments

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

Appendix E. Change Log

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

E.1. since draft-ietf-quic-load-balancers-09

*Renamed "Stream Cipher" and "Block Cipher" to "Encrypted Short" and "Encrypted Long"

*Changed "Encrypted Short" to a 4-pass algorithm.

*Recommended a random initial nonce for encrypted short.

E.2. since draft-ietf-quic-load-balancers-08

*Eliminate Dynamic SID allocation

*Eliminated server use bytes

E.3. since draft-ietf-quic-load-balancers-07

*Shortened SSCID nonce minimum length to 4 bytes

*Removed RSCID from Retry token body

*Simplified CID formats

*Shrunk size of SID table

E.4. since draft-ietf-quic-load-balancers-06

*Added interoperability with DTLS

*Changed "non-compliant" to "unroutable"

*Changed "arbitrary" algorithm to "fallback"

*Revised security considerations for mistrustful tenants

*Added retry service considerations for non-Initial packets

E.5. since draft-ietf-quic-load-balancers-05

*Added low-config CID for further discussion

*Complete revision of shared-state Retry Token

*Added YANG model

*Updated configuration limits to ensure CID entropy

*Switched to notation from quic-transport

E.6. since draft-ietf-quic-load-balancers-04

*Rearranged the shared-state retry token to simplify token processing

*More compact timestamp in shared-state retry token

*Revised server requirements for shared-state retries

*Eliminated zero padding from the test vectors

*Added server use bytes to the test vectors

*Additional compliant DCID criteria

E.7. since-draft-ietf-quic-load-balancers-03

*Improved Config Rotation text

*Added stream cipher test vectors

*Deleted the Obfuscated CID algorithm

E.8. since-draft-ietf-quic-load-balancers-02

*Replaced stream cipher algorithm with three-pass version

*Updated Retry format to encode info for required TPs

*Added discussion of version invariance

*Cleaned up text about config rotation

*Added Reset Oracle and limited configuration considerations

*Allow dropped long-header packets for known QUIC versions

E.9. since-draft-ietf-quic-load-balancers-01

*Test vectors for load balancer decoding

*Deleted remnants of in-band protocol

*Light edit of Retry Services section

*Discussed load balancer chains

E.10. since-draft-ietf-quic-load-balancers-00

* Removed in-band protocol from the document

E.11. Since draft-duke-quic-load-balancers-06

* Switch to IETF WG draft.

E.12. Since draft-duke-quic-load-balancers-05

* Editorial changes

* Made load balancer behavior independent of QUIC version

* Got rid of token in stream cipher encoding, because server might not have it

* Defined "non-compliant DCID" and specified rules for handling them.

* Added psuedocode for config schema

E.13. Since draft-duke-quic-load-balancers-04

* Added standard for retry services

E.14. Since draft-duke-quic-load-balancers-03

* Renamed Plaintext CID algorithm as Obfuscated CID

* Added new Plaintext CID algorithm

* Updated to allow 20B CIDs

* Added self-encoding of CID length

E.15. Since draft-duke-quic-load-balancers-02

* Added Config Rotation

* Added failover mode

* Tweaks to existing CID algorithms

* Added Block Cipher CID algorithm

* Reformatted QUIC-LB packets

E.16. Since draft-duke-quic-load-balancers-01

* Complete rewrite

*Supports multiple security levels

*Lightweight messages

E.17. 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