Workgroup: Network Working Group
Internet-Draft:
draft-ietf-quic-applicability-15
Published: 7 March 2022
Intended Status: Informational
Expires: 8 September 2022
Authors: M. Kuehlewind
Ericsson
B. Trammell
Google

Applicability of the QUIC Transport Protocol

Abstract

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

Status of This Memo

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

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

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

This Internet-Draft will expire on 8 September 2022.

Copyright Notice

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

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

Table of Contents

1. Introduction

QUIC [QUIC] is a new transport protocol providing a number of advanced features. While initially designed for the HTTP use case, it provides capabilities that can be used with a much wider variety of applications. QUIC is encapsulated in UDP. QUIC version 1 integrates TLS 1.3 [TLS13] to encrypt all payload data and most control information. The version of HTTP that uses QUIC is known as HTTP/3 [QUIC-HTTP].

This document provides guidance for application developers that want to use the QUIC protocol without implementing it on their own. This

includes general guidance for applications operating over HTTP/3 or directly over QUIC.

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

2. The Necessity of Fallback

QUIC uses UDP as a substrate. This enables userspace implementation and permits traversal of network middleboxes (including NAT) without requiring updates to existing network infrastructure.

Measurement studies have shown between three [Trammell16] and five [Swett16] percent of networks block all UDP traffic, though there is little evidence of other forms of systematic disadvantage to UDP traffic compared to TCP [Edeline16]. This blocking implies that all applications running on top of QUIC must either be prepared to accept connectivity failure on such networks, or be engineered to fall back to some other transport protocol. In the case of HTTP, this fallback is TLS over TCP.

The IETF TAPS specifications [I-D.ietf-taps-arch] describe a system with a common API for multiple protocols. This is particularly relevant for QUIC as it addresses the implications of fallback among multiple protocols.

Specifically, fallback to insecure protocols or to weaker versions of secure protocols needs to be avoided. In general, a application that implements fallback needs to consider the security consequences. A fallback to TCP and TLS exposes control information to modification and manipulation in the network. Further, downgrades to older TLS versions than 1.3, which is used in QUIC version 1, might result in significantly weaker cryptographic protection. For example, the results of protocol negotiation [RFC7301] only have confidentiality protection if TLS 1.3 is used.

These applications must operate, perhaps with impaired functionality, in the absence of features provided by QUIC not present in the fallback protocol. For fallback to TLS over TCP, the most obvious difference is that TCP does not provide stream multiplexing and therefore stream multiplexing would need to be implemented in the application layer if needed. Further, TCP implementations and network paths often do not support the Fast Open option [RFC7413], which enables sending of payload data together with the first control packet of a new connection as also provided by 0-RTT session resumption in QUIC. Note that there is some evidence of middleboxes blocking SYN data even if TFO was successfully negotiated (see [PaaschNanog]). And even if Fast Open

successfully operates end-to-end, it is limited to a single packet of TLS handshake and application data, unlike QUIC 0-RTT.

Moreover, while encryption (in this case TLS) is inseparably integrated with QUIC, TLS negotiation over TCP can be blocked. If TLS over TCP cannot be supported, the connection should be aborted, and the application then ought to present a suitable prompt to the user that secure communication is unavailable.

In summary, any fallback mechanism is likely to impose a degradation of performance and can degrade security; however, fallback must not silently violate the application's expectation of confidentiality or integrity of its payload data.

3. Zero RTT

QUIC provides for 0-RTT connection establishment. Though the same facility exists in TLS 1.3 with TCP, 0-RTT presents opportunities and challenges for applications using QUIC.

A transport protocol that provides 0-RTT connection establishment is qualitatively different than one that does not from the point of view of the application using it. Relative trade-offs between the cost of closing and reopening a connection and trying to keep it open are different; see Section 3.2.

An application needs to deliberately choose to use 0-RTT, as 0-RTT carries a risk of replay attack. Application protocols that use 0-RTT require a profile that describes the types of information that can be safely sent. For HTTP, this profile is described in [HTTP-REPLAY].

3.1. Replay Attacks

Retransmission or (malicious) replay of data contained in 0-RTT packets could cause the server side to receive multiple copies of the same data.

Application data sent by the client in 0-RTT packets could be processed more than once if it is replayed. Applications need to be aware of what is safe to send in 0-RTT. Application protocols that seek to enable the use of 0-RTT need a careful analysis and a description of what can be sent in 0-RTT; see Section 5.6 of [QUIC-TLS].

In some cases, it might be sufficient to limit application data sent in 0-RTT to that which only causes actions at a server that are known to be free of lasting effect. Initiating data retrieval or establishing configuration are examples of actions that could be safe. Idempotent operations - those for which repetition has the

same net effect as a single operation - might be safe. However, it is also possible to combine individually idempotent operations into a non-idempotent sequence of operations.

Once a server accepts 0-RTT data there is no means of selectively discarding data that is received. However, protocols can define ways to reject individual actions that might be unsafe if replayed.

Some TLS implementations and deployments might be able to provide partial or even complete replay protection, which could be used to manage replay risk.

3.2. Session resumption versus Keep-alive

Because QUIC is encapsulated in UDP, applications using QUIC must deal with short network idle timeouts. Deployed stateful middleboxes will generally establish state for UDP flows on the first packet sent, and keep state for much shorter idle periods than for TCP. [RFC5382] suggests a TCP idle period of at least 124 minutes, though there is no evidence of widespread implementation of this guideline in the literature. Short network timeout for UDP, however, is well-documented. According to a 2010 study ([Hatonen10]), UDP applications can assume that any NAT binding or other state entry can expire after just thirty seconds of inactivity. Section 3.5 of [RFC8085] further discusses keep-alive intervals for UDP: it requires a minimum value of 15 seconds, but recommends larger values, or omitting keep-alive entirely.

By using a connection ID, QUIC is designed to be robust to NAT address rebinding after a timeout. However, this only helps if one endpoint maintains availability at the address its peer uses, and the peer is the one to send after the timeout occurs.

Some QUIC connections might not be robust to NAT rebinding because the routing infrastructure (in particular, load balancers) uses the address/port four-tuple to direct traffic. Furthermore, middleboxes with functions other than address translation could still affect the path. In particular, some firewalls do not admit server traffic for which the firewall has no recent state for a corresponding packet sent from the client.

QUIC applications can adjust idle periods to manage the risk of timeout. Idle periods and the network idle timeout are distinct from the connection idle timeout, which is defined as the minimum of either endpoint's idle timeout parameter; see Section 10.1 of [QUIC]). There are three options:

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

*Ensure there are no long idle periods.

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

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

Either the server or the client in a QUIC application can send PING frames as keep-alives, to prevent the connection and any on-path state from timing out. Recommendations for the use of keep-alives are application-specific, mainly depending on the latency requirements and message frequency of the application. In this case, the application mapping must specify whether the client or server is responsible for keeping the application alive. While [Hatonen10] suggests that 30 seconds might be a suitable value for the public Internet when a NAT is on path, larger values are preferable if the deployment can consistently survive NAT rebinding or is known to be in a controlled environment (e.g. data centres) in order to lower network and computational load.

Sending PING frames more frequently than every 30 seconds over long idle periods may result in excessive unproductive traffic in some situations, and to unacceptable power usage for power-constrained (mobile) devices. Additionally, timeouts shorter than 30 seconds can make it harder to handle transient network interruptions, such as VM migration or coverage loss during mobilty. See [RFC8085], especially Section 3.5.

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

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

4. Use of Streams

QUIC's stream multiplexing feature allows applications to run multiple streams over a single connection, without head-of-line blocking between streams. Stream data is carried within frames,

where one QUIC packet on the wire can carry one or multiple stream frames.

Streams can be unidirectional or bidirectional, and a stream may be initiated either by client or server. Only the initiator of a unidirectional stream can send data on it.

Streams and connections can each carry a maximum of 2^{62}-1 bytes in each direction, due to encoding limitations on stream offsets and connection flow control limits. In the presently unlikely event that this limit is reached by an application, a new connection would need to be established.

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

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

Mapping of application data to streams is application-specific and described for HTTP/3 in [QUIC-HTTP]. There are a few general principles to apply when designing an application's use of streams:

* A single stream provides ordering. If the application requires certain data to be received in order, that data should be sent on the same stream. There is no guarantee of transmission, reception, or delivery order across streams.

* Multiple streams provide concurrency. Data that can be processed independently, and therefore would suffer from head of line blocking if forced to be received in order, should be transmitted over separate streams.

* Streams can provide message orientation, and allow messages to be cancelled. If one message is mapped to a single stream, resetting the stream to expire an unacknowledged message can be used to emulate partial reliability for that message.

If a QUIC receiver has opened the maximum allowed concurrent streams, and the sender indicates that more streams are needed, it does not automatically lead to an increase of the maximum number of

streams by the receiver. Therefore, an application can use the maximum number of allowed, currently open, and currently used streams when determining how to map data to streams.

QUIC assigns a numerical identifier to each stream, called the stream ID. While the relationship between these identifiers and stream types is clearly defined in version 1 of QUIC, future versions might change this relationship for various reasons. QUIC implementations should expose the properties of each stream (which endpoint initiated the stream, whether the stream is unidirectional or bidirectional, the stream ID used for the stream); applications should query for these properties rather than attempting to infer them from the stream ID.

The method of allocating stream identifiers to streams opened by the application might vary between transport implementations. Therefore, an application should not assume a particular stream ID will be assigned to a stream that has not yet been allocated. For example, HTTP/3 uses stream IDs to refer to streams that have already been opened, but makes no assumptions about future stream IDs or the way in which they are assigned Section 6 of [QUIC-HTTP]).

4.1. Stream versus Flow Multiplexing

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

4.2. Prioritization

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

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

4.3. Ordered and Reliable Delivery

QUIC streams enable ordered and reliable delivery. Though it is possible for an implementation to provide options that use streams for partial reliability or out-of-order delivery, most implementations will assume that data is reliably delivered in order.

Under this assumption, an endpoint that receives stream data might not make forward progress until data that is contiguous with the start of a stream is available. In particular, a receiver might withhold flow control credit until contiguous data is delivered to the application; see Section 2.2 of [QUIC]. To support this receive logic, an endpoint will send stream data until it is acknowledged, ensuring that data at the start of the stream is sent and acknowledged first.

An endpoint that uses a different sending behavior and does not negotiate that change with its peer might encounter performance issues or deadlocks.

4.4. Flow Control Deadlocks

QUIC flow control Section 4 of [QUIC] provides a means of managing access to the limited buffers endpoints have for incoming data. This mechanism limits the amount of data that can be in buffers in endpoints or in transit on the network. However, there are several ways in which limits can produce conditions that can cause a connection to either perform suboptimally or deadlock.

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

The size and rate of transport flow control credit updates can affect performance. Applications that use QUIC often have a data

consumer that reads data from transport buffers. Some implementations might have independent transport-layer and application-layer receive buffers. Consuming data does not always imply it is immediately processed. However, a common flow control implementation technique is to extend credit to the sender, by emitting MAX_DATA and/or MAX_STREAM_DATA frames, as data is consumed. Delivery of these frames is affected by the latency of the back channel from the receiver to the data sender. If credit is not extended in a timely manner, the sending application can be blocked, effectively throttling the sender.

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

A length-prefixed message format makes it easier for a data consumer to leave data unread in the transport buffer and thereby withhold flow control credit. If flow control limits prevent the remainder of a message from being sent, a deadlock will result. A length prefix might also enable the detection of this sort of deadlock. Where application protocols have messages that might be processed as a single unit, reserving flow control credit for the entire message atomically makes this style of deadlock less likely.

A data consumer can eagerly read all data as it becomes available, in order to make the receiver extend flow control credit and reduce the chances of a deadlock. However, such a data consumer might need other means for holding a peer accountable for the additional state it keeps for partially processed messages.

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

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

4.5. Stream Limit Commitments

QUIC endpoints are responsible for communicating the cumulative limit of streams they would allow to be opened by their peer. Initial limits are advertised using the initial_max_streams_bidi and initial_max_streams_uni transport parameters. As streams are opened and closed they are consumed and the cumulative total is incremented. Limits can be increased using the MAX_STREAMS frame but there is no mechanism to reduce limits. Once stream limits are reached, no more streams can be opened, which prevents applications using QUIC from making further progress. At this stage connections can be terminated via idle timeout or explicit close; see Section 10).

An application that uses QUIC and communicated a cumulative stream limit might require the connection to be closed before the limit is reached. For example, to stop the server to perform scheduled maintenance. Immediate connection close causes abrupt closure of actively used streams. Depending on how an application uses QUIC streams, this could be undesirable or detrimental to behavior or performance.

A more graceful closure technique is to stop sending increases to stream limits and allow the connection to naturally terminate once remaining streams are consumed. However, the period of time it takes to do so is dependent on the peer and an unpredictable closing period might not fit application or operational needs. Applications using QUIC can be conservative with open stream limits in order to reduce the commitment and indeterminism. However, being overly conservative with stream limits affects stream concurrency. Balancing these aspects can be specific to applications and their deployments.

Instead of relying on stream limits to avoid abrupt closure, an application-layer graceful close mechanism can be used to communicate the intention to explicitly close the connection at some future point. HTTP/3 provides such a mechanism using the GOAWAY frame. In HTTP/3, when the GOAWAY frame is received by a client, it stops opening new streams even if the cumulative stream limit would allow. Instead, the client would create a new connection on which to open further streams. Once all streams are closed on the old connection, it can be terminated safely by a connection close or after expiration of the idle time out (see also Section 10).

5. Packetization and Latency

QUIC exposes an interface that provides multiple streams to the application; however, the application usually cannot control how

data transmitted over those streams is mapped into frames or how those frames are bundled into packets.

By default, many implementations will try to maximally pack QUIC packets DATA frames from one or more streams to minimize bandwidth consumption and computational costs (see Section 13 of [QUIC]). If there is not enough data available to fill a packet, an implementation might wait for a short time, to optimize bandwidth efficiency instead of latency. This delay can either be pre-configured or dynamically adjusted based on the observed sending pattern of the application.

If the application requires low latency, with only small chunks of data to send, it may be valuable to indicate to QUIC that all data should be sent out immediately. Alternatively, if the application expects to use a specific sending pattern, it can also provide a suggested delay to QUIC for how long to wait before bundle frames into a packet.

Similarly, an application has usually no control about the length of a QUIC packet on the wire. QUIC provides the ability to add a PADDING frame to arbitrarily increase the size of packets. Padding is used by QUIC to ensure that the path is capable of transferring datagrams of at least a certain size, during the handshake (see Sections 8.1 and 14.1 of [QUIC]) and for path validation after connection migration (see Section 8.2 of [QUIC]) as well as for Datagram Packetization Layer PMTU Discovery (DPLMTUD) (see Section 14.3 of [QUIC]).

Padding can also be used by an application to reduce leakage of information about the data that is sent. A QUIC implementation can expose an interface that allows an application layer to specify how to apply padding.

6. Error Handling

QUIC recommends that endpoints signal any detected errors to the peer. Errors can occur at the transport level and the application level. Transport errors, such as a protocol violation, affect the entire connection. Applications that use QUIC can define their own error detection and signaling (see, for example, Section 8 of [QUIC-HTTP]). Application errors can affect an entire connection or a single stream.

QUIC defines an error code space that is used for error handling at the transport layer. QUIC encourages endpoints to use the most specific code, although any applicable code is permitted, including generic ones.

Applications using QUIC define an error code space that is independent from QUIC or other applications (see, for example, Section 8.1 of [QUIC-HTTP]). The values in an application error code space can be reused across connection-level and stream-level errors.

Connection errors lead to connection termination. They are signaled using a CONNECTION_CLOSE frame, which contains an error code and a reason field that can be zero length. Different types of CONNECTION_CLOSE frame are used to signal transport and application errors.

Stream errors lead to stream termination. These are signaled using STOP_SENDING or RESET_STREAM frames, which contain only an error code.

7. Acknowledgment Efficiency

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

8. Port Selection and Application Endpoint Discovery

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

As QUIC is a general-purpose transport protocol, there are no requirements that servers use a particular UDP port for QUIC. For applications with a fallback to TCP that do not already have an alternate mapping to UDP, usually the registration (if necessary) and use of the UDP port number corresponding to the TCP port already registered for the application is appropriate. For example, the default port for HTTP/3 [QUIC-HTTP] is UDP port 443, analogous to HTTP/1.1 or HTTP/2 over TLS over TCP.

Given the prevalence of the assumption in network management practice that a port number maps unambiguously to an application, the use of ports that cannot easily be mapped to a registered

service name might lead to blocking or other changes to the forwarding behavior by network elements such as firewalls that use the port number for application identification.

Applications could define an alternate endpoint discovery mechanism to allow the usage of ports other than the default. For example, HTTP/3 (Sections 3.2 and 3.3 of [QUIC-HTTP]) specifies the use of HTTP Alternative Services [RFC7838] for an HTTP origin to advertise the availability of an equivalent HTTP/3 endpoint on a certain UDP port by using the "h3" Application-Layer Protocol Negotiation (ALPN) [RFC7301] token.

ALPN permits the client and server to negotiate which of several protocols will be used on a given connection. Therefore, multiple applications might be supported on a single UDP port based on the ALPN token offered. Applications using QUIC are required to register an ALPN token for use in the TLS handshake.

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

8.1. Source Port Selection

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

* port 53 - DNS [RFC1034]

* port 123 - NTP [RFC5905]

* port 1900 - SSDP [SSDP]

* port 5353 - mDNS [RFC6762]

* port 11211 - memcached

Services might block source ports associated with protocols known to be vulnerable to reflection attacks, to avoid the overhead of processing large numbers of packets. However, this practice has negative effects on clients: not only does it require establishment of a new connection, but in some instances, might cause the client

to avoid using QUIC for that service for a period of time, downgrading to a non-UDP protocol (see Section 2).

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

9. Connection Migration

QUIC supports connection migration by the client. If an IP address changes, a QUIC endpoint can still associate packets with an existing transport connection using the Destination Connection ID field (see also Section 11) in the QUIC header. This supports cases where address information changes, such as NAT rebinding, intentional change of the local interface, or based on an indication in the handshake of the server for a preferred address to be used.

Use of a non-zero-length connection ID for the server is strongly recommended if any clients are behind a NAT or could be. A non-zero-length connection ID is also strongly recommended when active migration is supported. If a connection is intentionally migrated to new path, a new connection ID is used to minimize linkability by network observers. The other QUIC endpoint uses the connection ID to link different addresses to the same connection and entity if a non-zero-length connection ID is provided.

The base specification of QUIC version 1 only supports the use of a single network path at a time, which enables failover use cases. Path validation is required so that endpoints validate paths before use to avoid address spoofing attacks. Path validation takes at least one RTT and congestion control will also be reset after path migration. Therefore, migration usually has a performance impact.

QUIC probing packets, which can be sent on multiple paths at once, are used to perform address validation as well as measure path characteristics. Probing packets cannot carry application data but likely contain padding frames. Endpoints can use information about their receipt as input to congestion control for that path. Applications could use information learned from probing to inform a decision to switch paths.

Only the client can actively migrate in version 1 of QUIC. However, servers can indicate during the handshake that they prefer to transfer the connection to a different address after the handshake. For instance, this could be used to move from an address that is shared by multiple servers to an address that is unique to the server instance. The server can provide an IPv4 and an IPv6 address in a transport parameter during the TLS handshake and the client can

select between the two if both are provided. See also Section 9.6 of [QUIC].

10. Connection Termination

QUIC connections are terminated in one of three ways: implicit idle timeout, explicit immediate close, or explicit stateless reset.

QUIC does not provide any mechanism for graceful connection termination; applications using QUIC can define their own graceful termination process (see, for example, Section 5.2 of [QUIC-HTTP]).

QUIC idle timeout is enabled via transport parameters. Client and server announce a timeout period and the effective value for the connection is the minimum of the two values. After the timeout period elapses, the connection is silently closed. An application therefore should be able to configure its own maximum value, as well as have access to the computed minimum value for this connection. An application may adjust the maximum idle timeout for new connections based on the number of open or expected connections, since shorter timeout values may free-up resources more quickly.

Application data exchanged on streams or in datagrams defers the QUIC idle timeout. Applications that provide their own keep-alive mechanisms will therefore keep a QUIC connection alive. Applications that do not provide their own keep-alive can use transport-layer mechanisms (see Section 10.1.2 of [QUIC], and Section 3.2). However, QUIC implementation interfaces for controlling such transport behavior can vary, affecting the robustness of such approaches.

An immediate close is signaled by a CONNECTION_CLOSE frame (see Section 6). Immediate close causes all streams to become immediately closed, which may affect applications; see Section 4.5.

A stateless reset is an option of last resort for an endpoint that does not have access to connection state. Receiving a stateless reset is an indication of an unrecoverable error distinct from connection errors in that there is no application-layer information provided.

11. Information Exposure and the Connection ID

QUIC exposes some information to the network in the unencrypted part of the header, either before the encryption context is established or because the information is intended to be used by the network. For more information on manageability of QUIC see also [I-D.ietf-quic-manageability]. QUIC has a long header that exposes some additional information (the version and the source connection ID), while the short header exposes only the destination connection ID. In QUIC version 1, the long header is used during connection

establishment, while the short header is used for data transmission in an established connection.

The connection ID can be zero length. Zero length connection IDs can be chosen on each endpoint individually, on any packet except the first packets sent by clients during connection establishment.

An endpoint that selects a zero-length connection ID will receive packets with a zero-length destination connection ID. The endpoint needs to use other information, such as the source and destination IP address and port number to identify which connection is referred to. This could mean that the endpoint is unable to match datagrams to connections successfully if these values change, making the connection effectively unable to survive NAT rebinding or migrate to a new path.

11.1. Server-Generated Connection ID

QUIC supports a server-generated connection ID, transmitted to the client during connection establishment (see Section 7.2 of [QUIC]). Servers behind load balancers may need to change the connection ID during the handshake, encoding the identity of the server or information about its load balancing pool, in order to support stateless load balancing.

Server deployments with load balancers and other routing infrastructure need to ensure that this infrastructure consistently routes packets to the server instance that has the connection state, even if addresses, ports, and/or connection IDs change. This might require coordination between servers and infrastructure. One method of achieving this involves encoding routing information into the connection ID. For an example of this technique, see [QUIC-LB].

11.2. Mitigating Timing Linkability with Connection ID Migration

QUIC requires that endpoints generate fresh connection IDs for use on new network paths. Choosing values that are unlinkable to an outside observer ensures that activity on different paths cannot be trivially correlated using the connection ID.

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

In the limit where connection migration in a server pool is rare, it is trivial for an observer to associate two connection IDs. Conversely, in the opposite limit where every server handles

multiple simultaneous migrations, even an exposed server mapping may be insufficient information.

The most efficient mitigations for these attacks are through network design and/or operational practice, by using a load balancing architecture that loads more flows onto a single server-side address, by coordinating the timing of migrations in an attempt to increase the number of simultaneous migrations at a given time, or through other means.

11.3. Using Server Retry for Redirection

QUIC provides a Retry packet that can be sent by a server in response to the client Initial packet. The server may choose a new connection ID in that packet and the client will retry by sending another client Initial packet with the server-selected connection ID. This mechanism can be used to redirect a connection to a different server, e.g., due to performance reasons or when servers in a server pool are upgraded gradually, and therefore may support different versions of QUIC.

In this case, it is assumed that all servers belonging to a certain pool are served in cooperation with load balancers that forward the traffic based on the connection ID. A server can choose the connection ID in the Retry packet such that the load balancer will redirect the next Initial packet to a different server in that pool. Alternatively the load balancer can directly offer a Retry service as further described in [QUIC-LB].

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

12. Quality of Service (QoS) and DSCP

QUIC, as defined in [QUIC], has a single congestion controller and recovery handler. This design assumes that all packets of a QUIC connection, or at least with the same 5-tuple {dest addr, source addr, protocol, dest port, source port}, that have the same DiffServ Code Point (DSCP) [RFC2475] will receive similar network treatment since feedback about loss or delay of each packet is used as input to the congestion controller. Therefore, packets belonging to the same connection should use a single DSCP. Section 5.1 of [RFC7657] provides a discussion of DiffServ interactions with datagram transport protocols [RFC7657] (in this respect the interactions with QUIC resemble those of SCTP).

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

If differential network treatment is desired, e.g., by the use of different DSCPs, multiple QUIC connections to the same server may be used. However, in general it is recommended to minimize the number of QUIC connections to the same server, to avoid increased overhead and, more importantly, competing congestion control.

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

13. Use of Versions and Cryptographic Handshake

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

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

14. Enabling New Versions

QUIC version 1 does not specify a version negotation mechanism in the base spec but [I-D.draft-ietf-quic-version-negotiation] proposes an extension. This process assumes that the set of versions that a server supports is fixed. This complicates the process for deploying new QUIC versions or disabling old versions when servers operate in clusters.

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

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

compatible with an old version, because the new version is more likely to be used.

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

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

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

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

The third stage completes the process by enabling authentication of the negotiated version with the assumption that the new version is fully available.

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

15. Unreliable Datagram Service over QUIC

[I-D.ietf-quic-datagram] specifies a QUIC extension to enable sending and receiving unreliable datagrams over QUIC. Unlike operating directly over UDP, applications that use the QUIC datagram service do not need to implement their own congestion control, per [RFC8085], as QUIC datagrams are congestion controlled.

QUIC datagrams are not flow-controlled, and as such data chunks may be dropped if the receiver is overloaded. While the reliable transmission service of QUIC provides a stream-based interface to send and receive data in order over multiple QUIC streams, the datagram service has an unordered message-based interface. If

needed, an application layer framing can be used on top to allow separate flows of unreliable datagrams to be multiplexed on one QUIC connection.

16. IANA Considerations

This document has no actions for IANA; however, note that Section 8 recommends that application bindings to QUIC for applications using TCP register UDP ports analogous to their existing TCP registrations.

17. Security Considerations

See the security considerations in [QUIC] and [QUIC-TLS]; the security considerations for the underlying transport protocol are relevant for applications using QUIC, as well. Considerations on linkability, replay attacks, and randomness discussed in [QUIC-TLS] should be taken into account when deploying and using QUIC.

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

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

Further, [QUIC-HTTP] provides security considerations specific to HTTP. However, discussions such as on cross-protocol attacks, traffic analysis and padding, or migration might be relevant for other applications using QUIC as well.

18. Contributors

The following people have contributed significant text to and/or feedback on this document:

*Gorry Fairhurst

*Ian Swett

*Igor Lubashev

*Lucas Pardue

*Mike Bishop

*Mark Nottingham

*Martin Duke

*Martin Thomson

*Sean Turner

*Tommy Pauly

19. Acknowledgments

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

20. References

20.1. Normative References

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

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

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

20.2. Informative References

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

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

[HTTP-REPLAY] Thomson, M., Nottingham, M., and W. Tarreau, "Using Early Data in HTTP", RFC 8470, DOI 10.17487/RFC8470, September 2018, <https://www.rfc-editor.org/rfc/rfc8470>.

[I-D.draft-ietf-httpbis-priority]
Oku, K. and L. Pardue, "Extensible Prioritization Scheme for HTTP", Work in Progress, Internet-Draft, draft-ietf-httpbis-priority-12, 17 January 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-priority-12>.

[I-D.draft-ietf-quic-version-negotiation] Schinazi, D. and E. Rescorla, "Compatible Version Negotiation for QUIC", Work in Progress, Internet-Draft, draft-ietf-quic-version-negotiation-05, 25 October 2021, <https://datatracker.ietf.org/doc/html/draft-ietf-quic-version-negotiation-05>.

[I-D.ietf-quic-datagram] Pauly, T., Kinnear, E., and D. Schinazi, "An Unreliable Datagram Extension to QUIC", Work in Progress, Internet-Draft, draft-ietf-quic-datagram-10, 4 February 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-quic-datagram-10>.

[I-D.ietf-quic-manageability] Kuehlewind, M. and B. Trammell, "Manageability of the QUIC Transport Protocol", Work in Progress, Internet-Draft, draft-ietf-quic-manageability-14, 21 January 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-quic-manageability-14>.

[I-D.ietf-taps-arch] Pauly, T., Trammell, B., Brunstrom, A., Fairhurst, G., and C. Perkins, "An Architecture for Transport Services", Work in Progress, Internet-Draft, draft-ietf-taps-arch-12, 3 January 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-taps-arch-12>.

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

[QUIC-HTTP] Bishop, M., "Hypertext Transfer Protocol Version 3 (HTTP/3)", Work in Progress, Internet-Draft, draft-ietf-quic-http-34, 2 February 2021, <https://datatracker.ietf.org/doc/html/draft-ietf-quic-http-34>.

[QUIC-LB] Duke, M., Banks, N., and C. Huitema, "QUIC-LB: Generating Routable QUIC Connection IDs", Work in Progress, Internet-Draft, draft-ietf-quic-load-balancers-12, 11

February 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-quic-load-balancers-12>.

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

[RFC2475] Blake, S., Black, D., Carlson, M., Davies, E., Wang, Z., and W. Weiss, "An Architecture for Differentiated Services", RFC 2475, DOI 10.17487/RFC2475, December 1998, <https://www.rfc-editor.org/rfc/rfc2475>.

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

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

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

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

[RFC6762] Cheshire, S. and M. Krochmal, "Multicast DNS", RFC 6762, DOI 10.17487/RFC6762, February 2013, <https://www.rfc-editor.org/rfc/rfc6762>.

[RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/

RFC7301, July 2014, <https://www.rfc-editor.org/rfc/rfc7301>.

[RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <https://www.rfc-editor.org/rfc/rfc7413>.

[RFC7657] Black, D., Ed. and P. Jones, "Differentiated Services (Diffserv) and Real-Time Communication", RFC 7657, DOI 10.17487/RFC7657, November 2015, <https://www.rfc-editor.org/rfc/rfc7657>.

[RFC7838] Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", RFC 7838, DOI 10.17487/RFC7838, April 2016, <https://www.rfc-editor.org/rfc/rfc7838>.

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

[SSDP] Donoho, A., Roe, B., Bodlaender, M., Gildred, J., Messer, A., Kim, Y., Fairman, B., and J. Tourzan, "UPnP Device Architecture 2.0", 17 April 2020, <https://openconnectivity.org/upnp-specs/UPnP-arch-DeviceArchitecture-v2.0-20200417.pdf>.

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

[TLS13] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/rfc/rfc8446>.

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

Authors' Addresses

Mirja Kuehlewind
Ericsson

Email: mirja.kuehlewind@ericsson.com

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

Switzerland

Email: ietf@trammell.ch