Workgroup: QUIC
Internet-Draft: draft-ietf-quic-http-34
Published: 2 February 2021
Intended Status: Standards Track
Expires: 6 August 2021
Authors: M. Bishop, Ed.
Akamai

Hypertext Transfer Protocol Version 3 (HTTP/3)

Abstract

The QUIC transport protocol has several features that are desirable in a transport for HTTP, such as stream multiplexing, per-stream flow control, and low-latency connection establishment. This document describes a mapping of HTTP semantics over QUIC. This document also identifies HTTP/2 features that are subsumed by QUIC, and describes how HTTP/2 extensions can be ported to HTTP/3.

DO NOT DEPLOY THIS VERSION OF HTTP

DO NOT DEPLOY THIS VERSION OF HTTP/3 UNTIL IT IS IN AN RFC. This version is still a work in progress. For trial deployments, please use earlier versions.

Note to Readers

Discussion of this draft takes place on the QUIC working group mailing list (quic@ietf.org), which is archived at https://mailarchive.ietf.org/arch/search/?email_list=quic.

Working Group information can be found at https://github.com/quicwg; source code and issues list for this draft can be found at https://github.com/quicwg/base-drafts/labels/-http.

Status of This Memo

This Internet-Draft will expire on 6 August 2021.

Copyright Notice

Table of Contents

1. Introduction

HTTP semantics ([SEMANTICS]) are used for a broad range of services on the Internet. These semantics have most commonly been used with HTTP/1.1 and HTTP/2. HTTP/1.1 has been used over a variety of transport and session layers, while HTTP/2 has been used primarily with TLS over TCP. HTTP/3 supports the same semantics over a new transport protocol, QUIC.

1.1. Prior versions of HTTP

HTTP/1.1 ([HTTP11]) uses whitespace-delimited text fields to convey HTTP messages. While these exchanges are human-readable, using whitespace for message formatting leads to parsing complexity and excessive tolerance of variant behavior.

Because HTTP/1.1 does not include a multiplexing layer, multiple TCP connections are often used to service requests in parallel. However, that has a negative impact on congestion control and network efficiency, since TCP does not share congestion control across multiple connections.

HTTP/2 ([HTTP2]) introduced a binary framing and multiplexing layer to improve latency without modifying the transport layer. However, because the parallel nature of HTTP/2's multiplexing is not visible to TCP's loss recovery mechanisms, a lost or reordered packet causes all active transactions to experience a stall regardless of whether that transaction was directly impacted by the lost packet.

1.2. Delegation to QUIC

The QUIC transport protocol incorporates stream multiplexing and per-stream flow control, similar to that provided by the HTTP/2 framing layer. By providing reliability at the stream level and congestion control across the entire connection, QUIC has the capability to improve the performance of HTTP compared to a TCP mapping. QUIC also incorporates TLS 1.3 ([TLS13]) at the transport layer, offering comparable confidentiality and integrity to running TLS over TCP, with the improved connection setup latency of TCP Fast Open ([TFO]).

This document defines HTTP/3, a mapping of HTTP semantics over the QUIC transport protocol, drawing heavily on the design of HTTP/2. HTTP/3 relies on QUIC to provide confidentiality and integrity protection of data; peer authentication; and reliable, in-order, per-stream delivery. While delegating stream lifetime and flow control issues to QUIC, a binary framing similar to the HTTP/2 framing is used on each stream. Some HTTP/2 features are subsumed by QUIC, while other features are implemented atop QUIC.

QUIC is described in [QUIC-TRANSPORT]. For a full description of HTTP/2, see [HTTP2].

2. HTTP/3 Protocol Overview

HTTP/3 provides a transport for HTTP semantics using the QUIC transport protocol and an internal framing layer similar to HTTP/2.

Once a client knows that an HTTP/3 server exists at a certain endpoint, it opens a QUIC connection. QUIC provides protocol negotiation, stream-based multiplexing, and flow control. Discovery of an HTTP/3 endpoint is described in Section 3.1.

Within each stream, the basic unit of HTTP/3 communication is a frame (Section 7.2). Each frame type serves a different purpose. For example, HEADERS and DATA frames form the basis of HTTP requests and responses (Section 4.1). Frames that apply to the entire connection are conveyed on a dedicated control stream.

Multiplexing of requests is performed using the QUIC stream abstraction, described in Section 2 of [QUIC-TRANSPORT]. Each request-response pair consumes a single QUIC stream. Streams are independent of each other, so one stream that is blocked or suffers packet loss does not prevent progress on other streams.

Server push is an interaction mode introduced in HTTP/2 ([HTTP2]) that permits a server to push a request-response exchange to a client in anticipation of the client making the indicated request. This trades off network usage against a potential latency gain. Several HTTP/3 frames are used to manage server push, such as PUSH_PROMISE, MAX_PUSH_ID, and CANCEL_PUSH.

As in HTTP/2, request and response fields are compressed for transmission. Because HPACK ([HPACK]) relies on in-order transmission of compressed field sections (a guarantee not provided by QUIC), HTTP/3 replaces HPACK with QPACK ([QPACK]). QPACK uses separate unidirectional streams to modify and track field table state, while encoded field sections refer to the state of the table without modifying it.

2.1. Document Organization

The following sections provide a detailed overview of the lifecycle of an HTTP/3 connection:

* Connection Setup and Management (Section 3) covers how an HTTP/3 endpoint is discovered and an HTTP/3 connection is established.

* HTTP Request Lifecycle (Section 4) describes how HTTP semantics are expressed using frames.

* Connection Closure (Section 5) describes how HTTP/3 connections are terminated, either gracefully or abruptly.

The details of the wire protocol and interactions with the transport are described in subsequent sections:

* Stream Mapping and Usage (Section 6) describes the way QUIC streams are used.

* HTTP Framing Layer (Section 7) describes the frames used on most streams.

* Error Handling (Section 8) describes how error conditions are handled and expressed, either on a particular stream or for the connection as a whole.

Additional resources are provided in the final sections:

* Extensions to HTTP/3 (Section 9) describes how new capabilities can be added in future documents.

* A more detailed comparison between HTTP/2 and HTTP/3 can be found in Appendix A.

2.2. Conventions and Terminology

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

This document uses the variable-length integer encoding from [QUIC-TRANSPORT].

The following terms are used:

abort: An abrupt termination of a connection or stream, possibly due to an error condition.

client: The endpoint that initiates an HTTP/3 connection. Clients send HTTP requests and receive HTTP responses.

connection: A transport-layer connection between two endpoints, using QUIC as the transport protocol.

connection error: An error that affects the entire HTTP/3 connection.

endpoint: Either the client or server of the connection.

frame: The smallest unit of communication on a stream in HTTP/3, consisting of a header and a variable-length sequence of bytes structured according to the frame type.

Protocol elements called "frames" exist in both this document and [QUIC-TRANSPORT]. Where frames from [QUIC-TRANSPORT] are referenced, the frame name will be prefaced with "QUIC." For example, "QUIC CONNECTION_CLOSE frames." References without this preface refer to frames defined in Section 7.2.

HTTP/3 connection: A QUIC connection where the negotiated application protocol is HTTP/3.

peer: An endpoint. When discussing a particular endpoint, "peer" refers to the endpoint that is remote to the primary subject of discussion.

receiver: An endpoint that is receiving frames.

sender: An endpoint that is transmitting frames.

server: The endpoint that accepts an HTTP/3 connection. Servers receive HTTP requests and send HTTP responses.

stream: A bidirectional or unidirectional bytestream provided by the QUIC transport. All streams within an HTTP/3 connection can be considered "HTTP/3 streams," but multiple stream types are defined within HTTP/3.

stream error: An application-level error on the individual stream.

The term "content" is defined in Section 6.4 of [SEMANTICS].

Finally, the terms "resource", "message", "user agent", "origin server", "gateway", "intermediary", "proxy", and "tunnel" are defined in Section 3 of [SEMANTICS].

Packet diagrams in this document use the format defined in Section 1.3 of [QUIC-TRANSPORT] to illustrate the order and size of fields.

3. Connection Setup and Management

3.1. Discovering an HTTP/3 Endpoint

HTTP relies on the notion of an authoritative response: a response that has been determined to be the most appropriate response for that request given the state of the target resource at the time of response message origination by (or at the direction of) the origin server identified within the target URI. Locating an authoritative server for an HTTP URI is discussed in Section 4.3 of [SEMANTICS].

The "https" scheme associates authority with possession of a certificate that the client considers to be trustworthy for the host identified by the authority component of the URI. Upon receiving a server certificate in the TLS handshake, the client MUST verify that the certificate is an acceptable match for the URI's origin server using the process described in Section 4.3.4 of [SEMANTICS]. If the certificate cannot be verified with respect to the URI's origin server, the client MUST NOT consider the server authoritative for that origin.

A client MAY attempt access to a resource with an "https" URI by resolving the host identifier to an IP address, establishing a QUIC connection to that address on the indicated port (including validation of the server certificate as described above), and sending an HTTP/3 request message targeting the URI to the server over that secured connection. Unless some other mechanism is used to select HTTP/3, the token "h3" is used in the Application Layer Protocol Negotiation (ALPN; see [RFC7301]) extension during the TLS handshake.

Connectivity problems (e.g., blocking UDP) can result in QUIC connection establishment failure; clients SHOULD attempt to use TCP-based versions of HTTP in this case.

Servers MAY serve HTTP/3 on any UDP port; an alternative service advertisement always includes an explicit port, and URIs contain either an explicit port or a default port associated with the scheme.

3.1.1. HTTP Alternative Services

An HTTP origin can advertise the availability of an equivalent HTTP/3 endpoint via the Alt-Svc HTTP response header field or the HTTP/2 ALTSVC frame ([ALTSVC]), using the "h3" ALPN token.

For example, an origin could indicate in an HTTP response that HTTP/3 was available on UDP port 50781 at the same hostname by including the following header field:

```
Alt-Svc: h3=":50781"
```

On receipt of an Alt-Svc record indicating HTTP/3 support, a client MAY attempt to establish a QUIC connection to the indicated host and port; if this connection is successful, the client can send HTTP requests using the mapping described in this document.

3.1.2. Other Schemes

Although HTTP is independent of the transport protocol, the "http" scheme associates authority with the ability to receive TCP connections on the indicated port of whatever host is identified within the authority component. Because HTTP/3 does not use TCP, HTTP/3 cannot be used for direct access to the authoritative server for a resource identified by an "http" URI. However, protocol extensions such as [ALTSVC] permit the authoritative server to identify other services that are also authoritative and that might be reachable over HTTP/3.

Prior to making requests for an origin whose scheme is not "https", the client MUST ensure the server is willing to serve that scheme. For origins whose scheme is "http", an experimental method to accomplish this is described in [RFC8164]. Other mechanisms might be defined for various schemes in the future.

3.2. Connection Establishment

HTTP/3 relies on QUIC version 1 as the underlying transport. The use of other QUIC transport versions with HTTP/3 MAY be defined by future specifications.

QUIC version 1 uses TLS version 1.3 or greater as its handshake protocol. HTTP/3 clients MUST support a mechanism to indicate the target host to the server during the TLS handshake. If the server is identified by a domain name ([DNS-TERMS]), clients MUST send the Server Name Indication (SNI; [RFC6066]) TLS extension unless an alternative mechanism to indicate the target host is used.

QUIC connections are established as described in [QUIC-TRANSPORT]. During connection establishment, HTTP/3 support is indicated by selecting the ALPN token "h3" in the TLS handshake. Support for other application-layer protocols MAY be offered in the same handshake.

While connection-level options pertaining to the core QUIC protocol are set in the initial crypto handshake, HTTP/3-specific settings are conveyed in the SETTINGS frame. After the QUIC connection is established, a SETTINGS frame (Section 7.2.4) MUST be sent by each endpoint as the initial frame of their respective HTTP control stream; see Section 6.2.1.

3.3. Connection Reuse

HTTP/3 connections are persistent across multiple requests. For best performance, it is expected that clients will not close connections until it is determined that no further communication with a server is necessary (for example, when a user navigates away from a particular web page) or until the server closes the connection.

Once a connection exists to a server endpoint, this connection MAY be reused for requests with multiple different URI authority components. To use an existing connection for a new origin, clients MUST validate the certificate presented by the server for the new origin server using the process described in Section 4.3.4 of [SEMANTICS]. This implies that clients will need to retain the server certificate and any additional information needed to verify that certificate; clients which do not do so will be unable to reuse the connection for additional origins.

If the certificate is not acceptable with regard to the new origin for any reason, the connection MUST NOT be reused and a new connection SHOULD be established for the new origin. If the reason the certificate cannot be verified might apply to other origins already associated with the connection, the client SHOULD re-validate the server certificate for those origins. For instance, if validation of a certificate fails because the certificate has expired or been revoked, this might be used to invalidate all other origins for which that certificate was used to establish authority.

Clients SHOULD NOT open more than one HTTP/3 connection to a given IP address and UDP port, where the IP address and port might be derived from a URI, a selected alternative service ([ALTSVC]), a configured proxy, or name resolution of any of these. A client MAY open multiple HTTP/3 connections to the same IP address and UDP port using different transport or TLS configurations but SHOULD avoid creating multiple connections with the same configuration.

Servers are encouraged to maintain open HTTP/3 connections for as long as possible but are permitted to terminate idle connections if necessary. When either endpoint chooses to close the HTTP/3 connection, the terminating endpoint SHOULD first send a GOAWAY frame (Section 5.2) so that both endpoints can reliably determine whether previously sent frames have been processed and gracefully complete or terminate any necessary remaining tasks.

A server that does not wish clients to reuse HTTP/3 connections for a particular origin can indicate that it is not authoritative for a request by sending a 421 (Misdirected Request) status code in response to the request; see Section 7.4 of [SEMANTICS].

4. HTTP Request Lifecycle

4.1. HTTP Message Exchanges

A client sends an HTTP request on a request stream, which is a client-initiated bidirectional QUIC stream; see Section 6.1. A client MUST send only a single request on a given stream. A server sends zero or more interim HTTP responses on the same stream as the request, followed by a single final HTTP response, as detailed below. See Section 15 of [SEMANTICS] for a description of interim and final HTTP responses.

Pushed responses are sent on a server-initiated unidirectional QUIC stream; see Section 6.2.2. A server sends zero or more interim HTTP responses, followed by a single final HTTP response, in the same manner as a standard response. Push is described in more detail in Section 4.4.

On a given stream, receipt of multiple requests or receipt of an additional HTTP response following a final HTTP response MUST be treated as malformed (Section 4.1.3).

An HTTP message (request or response) consists of:

1. the header section, sent as a single HEADERS frame (see Section 7.2.2),

2. optionally, the content, if present, sent as a series of DATA frames (see Section 7.2.1), and

3. optionally, the trailer section, if present, sent as a single HEADERS frame.

Header and trailer sections are described in Sections 6.3 and 6.5 of [SEMANTICS]; the content is described in Section 6.4 of [SEMANTICS].

Receipt of an invalid sequence of frames MUST be treated as a connection error of type H3_FRAME_UNEXPECTED; see Section 8. In particular, a DATA frame before any HEADERS frame, or a HEADERS or

DATA frame after the trailing HEADERS frame, is considered invalid. Other frame types, especially unknown frame types, might be permitted subject to their own rules; see Section 9.

A server MAY send one or more PUSH_PROMISE frames (Section 7.2.5) before, after, or interleaved with the frames of a response message. These PUSH_PROMISE frames are not part of the response; see Section 4.4 for more details. PUSH_PROMISE frames are not permitted on push streams; a pushed response that includes PUSH_PROMISE frames MUST be treated as a connection error of type H3_FRAME_UNEXPECTED; see Section 8.

Frames of unknown types (Section 9), including reserved frames (Section 7.2.8) MAY be sent on a request or push stream before, after, or interleaved with other frames described in this section.

The HEADERS and PUSH_PROMISE frames might reference updates to the QPACK dynamic table. While these updates are not directly part of the message exchange, they must be received and processed before the message can be consumed. See Section 4.1.1 for more details.

Transfer codings (see Section 6.1 of [HTTP11]) are not defined for HTTP/3; the Transfer-Encoding header field MUST NOT be used.

A response MAY consist of multiple messages when and only when one or more interim responses (1xx; see Section 15.2 of [SEMANTICS]) precede a final response to the same request. Interim responses do not contain content or trailer sections.

An HTTP request/response exchange fully consumes a client-initiated bidirectional QUIC stream. After sending a request, a client MUST close the stream for sending. Unless using the CONNECT method (see Section 4.2), clients MUST NOT make stream closure dependent on receiving a response to their request. After sending a final response, the server MUST close the stream for sending. At this point, the QUIC stream is fully closed.

When a stream is closed, this indicates the end of the final HTTP message. Because some messages are large or unbounded, endpoints SHOULD begin processing partial HTTP messages once enough of the message has been received to make progress. If a client-initiated stream terminates without enough of the HTTP message to provide a complete response, the server SHOULD abort its response stream with the error code H3_REQUEST_INCOMPLETE; see Section 8.

A server can send a complete response prior to the client sending an entire request if the response does not depend on any portion of the request that has not been sent and received. When the server does not need to receive the remainder of the request, it MAY abort reading the request stream, send a complete response, and cleanly close the sending part of the stream. The error code H3_NO_ERROR SHOULD be used when requesting that the client stop sending on the request stream. Clients MUST NOT discard complete responses as a result of having their request terminated abruptly, though clients can always discard responses at their discretion for other reasons. If the server sends a partial or complete response but does not abort reading the request, clients SHOULD continue sending the body of the request and close the stream normally.

4.1.1. Field Formatting and Compression

HTTP messages carry metadata as a series of key-value pairs called HTTP fields; see Sections 6.3 and 6.5 of [SEMANTICS]. For a listing of registered HTTP fields, see the "Hypertext Transfer Protocol (HTTP) Field Name Registry" maintained at https://www.iana.org/assignments/http-fields/.

> **Note:** This registry will not exist until [SEMANTICS] is approved. **RFC Editor**, please remove this note prior to publication.

Field names are strings containing a subset of ASCII characters. Properties of HTTP field names and values are discussed in more detail in Section 5.1 of [SEMANTICS]. As in HTTP/2, characters in field names MUST be converted to lowercase prior to their encoding. A request or response containing uppercase characters in field names MUST be treated as malformed (Section 4.1.3).

Like HTTP/2, HTTP/3 does not use the Connection header field to indicate connection-specific fields; in this protocol, connection-specific metadata is conveyed by other means. An endpoint MUST NOT generate an HTTP/3 field section containing connection-specific fields; any message containing connection-specific fields MUST be treated as malformed (Section 4.1.3).

The only exception to this is the TE header field, which MAY be present in an HTTP/3 request header; when it is, it MUST NOT contain any value other than "trailers".

An intermediary transforming an HTTP/1.x message to HTTP/3 MUST remove connection-specific header fields as discussed in Section 7.6.1 of [SEMANTICS], or their messages will be treated by other HTTP/3 endpoints as malformed (Section 4.1.3).

4.1.1.1. Pseudo-Header Fields

Like HTTP/2, HTTP/3 employs a series of pseudo-header fields where the field name begins with the ':' character (ASCII 0x3a). These pseudo-header fields convey the target URI, the method of the request, and the status code for the response.

Pseudo-header fields are not HTTP fields. Endpoints MUST NOT generate pseudo-header fields other than those defined in this document; however, an extension could negotiate a modification of this restriction; see Section 9.

Pseudo-header fields are only valid in the context in which they are defined. Pseudo-header fields defined for requests MUST NOT appear in responses; pseudo-header fields defined for responses MUST NOT appear in requests. Pseudo-header fields MUST NOT appear in trailer sections. Endpoints MUST treat a request or response that contains undefined or invalid pseudo-header fields as malformed (Section 4.1.3).

All pseudo-header fields MUST appear in the header section before regular header fields. Any request or response that contains a pseudo-header field that appears in a header section after a regular header field MUST be treated as malformed (Section 4.1.3).

The following pseudo-header fields are defined for requests:

":method": Contains the HTTP method (Section 9 of [SEMANTICS])

":scheme": Contains the scheme portion of the target URI (Section 3.1 of [URI])

":scheme" is not restricted to URIs with scheme "http" and "https". A proxy or gateway can translate requests for non-HTTP schemes, enabling the use of HTTP to interact with non-HTTP services.

See Section 3.1.2 for guidance on using a scheme other than "https".

":authority": Contains the authority portion of the target URI (Section 3.2 of [URI]). The authority MUST NOT include the deprecated "userinfo" subcomponent for URIs of scheme "http" or "https".

To ensure that the HTTP/1.1 request line can be reproduced accurately, this pseudo-header field MUST be omitted when translating from an HTTP/1.1 request that has a request target in origin or asterisk form; see Section 7.1 of [SEMANTICS]. Clients that generate HTTP/3 requests directly SHOULD use the ":authority" pseudo-header field instead of the Host field. An intermediary that converts an HTTP/3 request to HTTP/1.1 MUST create a Host field if one is not present in a request by copying the value of the ":authority" pseudo-header field.

":path": Contains the path and query parts of the target URI (the "path-absolute" production and optionally a '?' character followed by the "query" production; see Sections 3.3 and 3.4 of [URI]. A request in asterisk form includes the value '*' for the ":path" pseudo-header field.

This pseudo-header field MUST NOT be empty for "http" or "https" URIs; "http" or "https" URIs that do not contain a path component MUST include a value of '/'. The exception to this rule is an OPTIONS request for an "http" or "https" URI that does not include a path component; these MUST include a ":path" pseudo-header field with a value of '*'; see Section 7.1 of [SEMANTICS].

All HTTP/3 requests MUST include exactly one value for the ":method", ":scheme", and ":path" pseudo-header fields, unless it is a CONNECT request; see Section 4.2.

If the ":scheme" pseudo-header field identifies a scheme that has a mandatory authority component (including "http" and "https"), the request MUST contain either an ":authority" pseudo-header field or a "Host" header field. If these fields are present, they MUST NOT be empty. If both fields are present, they MUST contain the same value. If the scheme does not have a mandatory authority component and none is provided in the request target, the request MUST NOT contain the ":authority" pseudo-header or "Host" header fields.

An HTTP request that omits mandatory pseudo-header fields or contains invalid values for those pseudo-header fields is malformed (Section 4.1.3).

HTTP/3 does not define a way to carry the version identifier that is included in the HTTP/1.1 request line.

For responses, a single ":status" pseudo-header field is defined that carries the HTTP status code; see Section 15 of [SEMANTICS]. This pseudo-header field MUST be included in all responses; otherwise, the response is malformed (Section 4.1.3).

HTTP/3 does not define a way to carry the version or reason phrase that is included in an HTTP/1.1 status line.

4.1.1.2. Field Compression

[QPACK] describes a variation of HPACK that gives an encoder some control over how much head-of-line blocking can be caused by compression. This allows an encoder to balance compression efficiency with latency. HTTP/3 uses QPACK to compress header and trailer sections, including the pseudo-header fields present in the header section.

To allow for better compression efficiency, the "Cookie" field ([RFC6265]) MAY be split into separate field lines, each with one or more cookie-pairs, before compression. If a decompressed field section contains multiple cookie field lines, these MUST be concatenated into a single byte string using the two-byte delimiter of 0x3b, 0x20 (the ASCII string "; ") before being passed into a context other than HTTP/2 or HTTP/3, such as an HTTP/1.1 connection, or a generic HTTP server application.

4.1.1.3. Header Size Constraints

An HTTP/3 implementation MAY impose a limit on the maximum size of the message header it will accept on an individual HTTP message. A server that receives a larger header section than it is willing to handle can send an HTTP 431 (Request Header Fields Too Large) status code ([RFC6585]). A client can discard responses that it cannot process. The size of a field list is calculated based on the uncompressed size of fields, including the length of the name and value in bytes plus an overhead of 32 bytes for each field.

If an implementation wishes to advise its peer of this limit, it can be conveyed as a number of bytes in the SETTINGS_MAX_FIELD_SECTION_SIZE parameter. An implementation that has received this parameter SHOULD NOT send an HTTP message header that exceeds the indicated size, as the peer will likely refuse to process it. However, an HTTP message can traverse one or more intermediaries before reaching the origin server; see Section 3.7 of [SEMANTICS]. Because this limit is applied separately by each implementation which processes the message, messages below this limit are not guaranteed to be accepted.

4.1.2. Request Cancellation and Rejection

Once a request stream has been opened, the request MAY be cancelled by either endpoint. Clients cancel requests if the response is no longer of interest; servers cancel requests if they are unable to or choose not to respond. When possible, it is RECOMMENDED that servers send an HTTP response with an appropriate status code rather than canceling a request it has already begun processing.

Implementations SHOULD cancel requests by abruptly terminating any directions of a stream that are still open. This means resetting the sending parts of streams and aborting reading on receiving parts of streams; see Section 2.4 of [QUIC-TRANSPORT].

When the server cancels a request without performing any application processing, the request is considered "rejected." The server SHOULD abort its response stream with the error code H3_REQUEST_REJECTED. In this context, "processed" means that some data from the stream was passed to some higher layer of software that might have taken some action as a result. The client can treat requests rejected by the server as though they had never been sent at all, thereby allowing them to be retried later.

Servers MUST NOT use the H3_REQUEST_REJECTED error code for requests that were partially or fully processed. When a server abandons a response after partial processing, it SHOULD abort its response stream with the error code H3_REQUEST_CANCELLED.

Client SHOULD use the error code H3_REQUEST_CANCELLED to cancel requests. Upon receipt of this error code, a server MAY abruptly terminate the response using the error code H3_REQUEST_REJECTED if no processing was performed. Clients MUST NOT use the H3_REQUEST_REJECTED error code, except when a server has requested closure of the request stream with this error code.

If a stream is canceled after receiving a complete response, the client MAY ignore the cancellation and use the response. However, if a stream is cancelled after receiving a partial response, the response SHOULD NOT be used. Only idempotent actions such as GET, PUT, or DELETE can be safely retried; a client SHOULD NOT automatically retry a request with a non-idempotent method unless it has some means to know that the request semantics are idempotent independent of the method or some means to detect that the original request was never applied. See Section 9.2.2 of [SEMANTICS] for more details.

4.1.3. Malformed Requests and Responses

A malformed request or response is one that is an otherwise valid sequence of frames but is invalid due to:

* the presence of prohibited fields or pseudo-header fields,

* the absence of mandatory pseudo-header fields,

* invalid values for pseudo-header fields,

* pseudo-header fields after fields,

* an invalid sequence of HTTP messages,

* the inclusion of uppercase field names, or

* the inclusion of invalid characters in field names or values.

A request or response that is defined as having content when it contains a Content-Length header field (Section 6.4.1 of [SEMANTICS]), is malformed if the value of a Content-Length header field does not equal the sum of the DATA frame lengths received. A response that is defined as never having content, even when a

Content-Length is present, can have a non-zero Content-Length field even though no content is included in DATA frames.

Intermediaries that process HTTP requests or responses (i.e., any intermediary not acting as a tunnel) MUST NOT forward a malformed request or response. Malformed requests or responses that are detected MUST be treated as a stream error (Section 8) of type H3_MESSAGE_ERROR.

For malformed requests, a server MAY send an HTTP response indicating the error prior to closing or resetting the stream. Clients MUST NOT accept a malformed response. Note that these requirements are intended to protect against several types of common attacks against HTTP; they are deliberately strict because being permissive can expose implementations to these vulnerabilities.

4.2. The CONNECT Method

The CONNECT method requests that the recipient establish a tunnel to the destination origin server identified by the request-target; see Section 9.3.6 of [SEMANTICS]. It is primarily used with HTTP proxies to establish a TLS session with an origin server for the purposes of interacting with "https" resources.

In HTTP/1.x, CONNECT is used to convert an entire HTTP connection into a tunnel to a remote host. In HTTP/2 and HTTP/3, the CONNECT method is used to establish a tunnel over a single stream.

A CONNECT request MUST be constructed as follows:

* The ":method" pseudo-header field is set to "CONNECT"

* The ":scheme" and ":path" pseudo-header fields are omitted

* The ":authority" pseudo-header field contains the host and port to connect to (equivalent to the authority-form of the request-target of CONNECT requests; see Section 7.1 of [SEMANTICS])

The request stream remains open at the end of the request to carry the data to be transferred. A CONNECT request that does not conform to these restrictions is malformed; see Section 4.1.3.

A proxy that supports CONNECT establishes a TCP connection ([RFC0793]) to the server identified in the ":authority" pseudo-header field. Once this connection is successfully established, the proxy sends a HEADERS frame containing a 2xx series status code to the client, as defined in Section 15.3 of [SEMANTICS].

All DATA frames on the stream correspond to data sent or received on the TCP connection. The payload of any DATA frame sent by the client is transmitted by the proxy to the TCP server; data received from the TCP server is packaged into DATA frames by the proxy. Note that the size and number of TCP segments is not guaranteed to map predictably to the size and number of HTTP DATA or QUIC STREAM frames.

Once the CONNECT method has completed, only DATA frames are permitted to be sent on the stream. Extension frames MAY be used if specifically permitted by the definition of the extension. Receipt of any other known frame type MUST be treated as a connection error of type H3_FRAME_UNEXPECTED; see Section 8.

The TCP connection can be closed by either peer. When the client ends the request stream (that is, the receive stream at the proxy enters the "Data Recvd" state), the proxy will set the FIN bit on its connection to the TCP server. When the proxy receives a packet with the FIN bit set, it will close the send stream that it sends to the client. TCP connections that remain half-closed in a single direction are not invalid, but are often handled poorly by servers, so clients SHOULD NOT close a stream for sending while they still expect to receive data from the target of the CONNECT.

A TCP connection error is signaled by abruptly terminating the stream. A proxy treats any error in the TCP connection, which includes receiving a TCP segment with the RST bit set, as a stream error of type H3_CONNECT_ERROR; see Section 8. Correspondingly, if a proxy detects an error with the stream or the QUIC connection, it MUST close the TCP connection. If the underlying TCP implementation permits it, the proxy SHOULD send a TCP segment with the RST bit set.

Since CONNECT creates a tunnel to an arbitrary server, proxies that support CONNECT SHOULD restrict its use to a set of known ports or a list of safe request targets; see Section 9.3.6 of [SEMANTICS] for more detail.

4.3. HTTP Upgrade

HTTP/3 does not support the HTTP Upgrade mechanism (Section 7.8 of [SEMANTICS]) or 101 (Switching Protocols) informational status code (Section 15.2.2 of [SEMANTICS]).

4.4. Server Push

Server push is an interaction mode that permits a server to push a request-response exchange to a client in anticipation of the client making the indicated request. This trades off network usage against a potential latency gain. HTTP/3 server push is similar to what is described in Section 8.2 of [HTTP2], but uses different mechanisms.

Each server push is assigned a unique Push ID by the server. The Push ID is used to refer to the push in various contexts throughout the lifetime of the HTTP/3 connection.

The Push ID space begins at zero, and ends at a maximum value set by the MAX_PUSH_ID frame; see Section 7.2.7. In particular, a server is not able to push until after the client sends a MAX_PUSH_ID frame. A client sends MAX_PUSH_ID frames to control the number of pushes that a server can promise. A server SHOULD use Push IDs sequentially, beginning from zero. A client MUST treat receipt of a push stream as a connection error of type H3_ID_ERROR (Section 8) when no MAX_PUSH_ID frame has been sent or when the stream references a Push ID that is greater than the maximum Push ID.

The Push ID is used in one or more PUSH_PROMISE frames (Section 7.2.5) that carry the header section of the request message. These frames are sent on the request stream that generated the push. This allows the server push to be associated with a client request. When the same Push ID is promised on multiple request streams, the decompressed request field sections MUST contain the same fields in the same order, and both the name and the value in each field MUST be identical.

The Push ID is then included with the push stream that ultimately fulfills those promises; see Section 6.2.2. The push stream identifies the Push ID of the promise that it fulfills, then contains a response to the promised request as described in Section 4.1.

Finally, the Push ID can be used in CANCEL_PUSH frames; see Section 7.2.3. Clients use this frame to indicate they do not wish to receive a promised resource. Servers use this frame to indicate they will not be fulfilling a previous promise.

Not all requests can be pushed. A server MAY push requests that have the following properties:

* cacheable; see Section 9.2.3 of [SEMANTICS]

* safe; see Section 9.2.1 of [SEMANTICS]

* does not include a request body or trailer section

The server MUST include a value in the ":authority" pseudo-header field for which the server is authoritative. If the client has not yet validated the connection for the origin indicated by the pushed request, it MUST perform the same verification process it would do before sending a request for that origin on the connection; see Section 3.3. If this verification fails, the client MUST NOT consider the server authoritative for that origin.

Clients SHOULD send a CANCEL_PUSH frame upon receipt of a PUSH_PROMISE frame carrying a request that is not cacheable, is not known to be safe, that indicates the presence of a request body, or for which it does not consider the server authoritative. Any corresponding responses MUST NOT be used or cached.

Each pushed response is associated with one or more client requests. The push is associated with the request stream on which the PUSH_PROMISE frame was received. The same server push can be associated with additional client requests using a PUSH_PROMISE frame with the same Push ID on multiple request streams. These associations do not affect the operation of the protocol, but MAY be considered by user agents when deciding how to use pushed resources.

Ordering of a PUSH_PROMISE frame in relation to certain parts of the response is important. The server SHOULD send PUSH_PROMISE frames prior to sending HEADERS or DATA frames that reference the promised responses. This reduces the chance that a client requests a resource that will be pushed by the server.

Due to reordering, push stream data can arrive before the corresponding PUSH_PROMISE frame. When a client receives a new push stream with an as-yet-unknown Push ID, both the associated client request and the pushed request header fields are unknown. The client can buffer the stream data in expectation of the matching PUSH_PROMISE. The client can use stream flow control (see Section 4.1 of [QUIC-TRANSPORT]) to limit the amount of data a server may commit to the pushed stream.

Push stream data can also arrive after a client has canceled a push. In this case, the client can abort reading the stream with an error code of H3_REQUEST_CANCELLED. This asks the server not to transfer additional data and indicates that it will be discarded upon receipt.

Pushed responses that are cacheable (see Section 3 of [CACHING]) can be stored by the client, if it implements an HTTP cache. Pushed responses are considered successfully validated on the origin server (e.g., if the "no-cache" cache response directive is present; see Section 5.2.2.3 of [CACHING]) at the time the pushed response is received.

Pushed responses that are not cacheable MUST NOT be stored by any HTTP cache. They MAY be made available to the application separately.

5. Connection Closure

Once established, an HTTP/3 connection can be used for many requests and responses over time until the connection is closed. Connection closure can happen in any of several different ways.

5.1. Idle Connections

Each QUIC endpoint declares an idle timeout during the handshake. If the QUIC connection remains idle (no packets received) for longer than this duration, the peer will assume that the connection has been closed. HTTP/3 implementations will need to open a new HTTP/3 connection for new requests if the existing connection has been idle for longer than the idle timeout negotiated during the QUIC handshake, and SHOULD do so if approaching the idle timeout; see Section 10.1 of [QUIC-TRANSPORT].

HTTP clients are expected to request that the transport keep connections open while there are responses outstanding for requests or server pushes, as described in Section 10.1.2 of [QUIC-TRANSPORT]. If the client is not expecting a response from the server, allowing an idle connection to time out is preferred over expending effort maintaining a connection that might not be needed. A gateway MAY maintain connections in anticipation of need rather than incur the latency cost of connection establishment to servers. Servers SHOULD NOT actively keep connections open.

5.2. Connection Shutdown

Even when a connection is not idle, either endpoint can decide to stop using the connection and initiate a graceful connection close. Endpoints initiate the graceful shutdown of an HTTP/3 connection by sending a GOAWAY frame (Section 7.2.6). The GOAWAY frame contains an identifier that indicates to the receiver the range of requests or pushes that were or might be processed in this connection. The server sends a client-initiated bidirectional Stream ID; the client sends a Push ID (Section 4.4). Requests or pushes with the indicated identifier or greater are rejected (Section 4.1.2) by the sender of the GOAWAY. This identifier MAY be zero if no requests or pushes were processed.

The information in the GOAWAY frame enables a client and server to agree on which requests or pushes were accepted prior to the shutdown of the HTTP/3 connection. Upon sending a GOAWAY frame, the endpoint SHOULD explicitly cancel (see Section 4.1.2 and Section 7.2.3) any requests or pushes that have identifiers greater than or equal to that indicated, in order to clean up transport state for the affected streams. The endpoint SHOULD continue to do so as more requests or pushes arrive.

Endpoints MUST NOT initiate new requests or promise new pushes on the connection after receipt of a GOAWAY frame from the peer. Clients MAY establish a new connection to send additional requests.

Some requests or pushes might already be in transit:

* Upon receipt of a GOAWAY frame, if the client has already sent requests with a Stream ID greater than or equal to the identifier contained in the GOAWAY frame, those requests will not be processed. Clients can safely retry unprocessed requests on a different HTTP connection. A client that is unable to retry requests loses all requests that are in flight when the server closes the connection.

  Requests on Stream IDs less than the Stream ID in a GOAWAY frame from the server might have been processed; their status cannot be known until a response is received, the stream is reset individually, another GOAWAY is received with a lower Stream ID than that of the request in question, or the connection terminates.

  Servers MAY reject individual requests on streams below the indicated ID if these requests were not processed.

* If a server receives a GOAWAY frame after having promised pushes with a Push ID greater than or equal to the identifier contained in the GOAWAY frame, those pushes will not be accepted.

Servers SHOULD send a GOAWAY frame when the closing of a connection is known in advance, even if the advance notice is small, so that the remote peer can know whether a request has been partially processed or not. For example, if an HTTP client sends a POST at the same time that a server closes a QUIC connection, the client cannot know if the server started to process that POST request if the server does not send a GOAWAY frame to indicate what streams it might have acted on.

An endpoint MAY send multiple GOAWAY frames indicating different identifiers, but the identifier in each frame MUST NOT be greater than the identifier in any previous frame, since clients might already have retried unprocessed requests on another HTTP connection. Receiving a GOAWAY containing a larger identifier than previously received MUST be treated as a connection error of type H3_ID_ERROR; see Section 8.

An endpoint that is attempting to gracefully shut down a connection can send a GOAWAY frame with a value set to the maximum possible value (2^{62}-4 for servers, 2^{62}-1 for clients). This ensures that the peer stops creating new requests or pushes. After allowing time for any in-flight requests or pushes to arrive, the endpoint can send another GOAWAY frame indicating which requests or pushes it might accept before the end of the connection. This ensures that a connection can be cleanly shut down without losing requests.

A client has more flexibility in the value it chooses for the Push ID in a GOAWAY that it sends. A value of 2^{62}-1 indicates that the server can continue fulfilling pushes that have already been promised. A smaller value indicates the client will reject pushes with Push IDs greater than or equal to this value. Like the server, the client MAY send subsequent GOAWAY frames so long as the specified Push ID is no greater than any previously sent value.

Even when a GOAWAY indicates that a given request or push will not be processed or accepted upon receipt, the underlying transport resources still exist. The endpoint that initiated these requests can cancel them to clean up transport state.

Once all accepted requests and pushes have been processed, the endpoint can permit the connection to become idle, or MAY initiate an immediate closure of the connection. An endpoint that completes a graceful shutdown SHOULD use the H3_NO_ERROR error code when closing the connection.

If a client has consumed all available bidirectional stream IDs with requests, the server need not send a GOAWAY frame, since the client is unable to make further requests.

5.3. Immediate Application Closure

An HTTP/3 implementation can immediately close the QUIC connection at any time. This results in sending a QUIC CONNECTION_CLOSE frame to the peer indicating that the application layer has terminated the connection. The application error code in this frame indicates to the peer why the connection is being closed. See Section 8 for error codes that can be used when closing a connection in HTTP/3.

Before closing the connection, a GOAWAY frame MAY be sent to allow the client to retry some requests. Including the GOAWAY frame in the same packet as the QUIC CONNECTION_CLOSE frame improves the chances of the frame being received by clients.

If there are open streams that have not been explicitly closed, they are implicitly closed when the connection is closed; see Section 10.2 of [QUIC-TRANSPORT].

5.4. Transport Closure

For various reasons, the QUIC transport could indicate to the application layer that the connection has terminated. This might be due to an explicit closure by the peer, a transport-level error, or a change in network topology that interrupts connectivity.

If a connection terminates without a GOAWAY frame, clients MUST assume that any request that was sent, whether in whole or in part, might have been processed.

6. Stream Mapping and Usage

A QUIC stream provides reliable in-order delivery of bytes, but makes no guarantees about order of delivery with regard to bytes on other streams. In version 1 of QUIC, the stream data containing HTTP frames is carried by QUIC STREAM frames, but this framing is invisible to the HTTP framing layer. The transport layer buffers and orders received stream data, exposing a reliable byte stream to the application. Although QUIC permits out-of-order delivery within a stream, HTTP/3 does not make use of this feature.

QUIC streams can be either unidirectional, carrying data only from initiator to receiver, or bidirectional. Streams can be initiated by either the client or the server. For more detail on QUIC streams, see Section 2 of [QUIC-TRANSPORT].

When HTTP fields and data are sent over QUIC, the QUIC layer handles most of the stream management. HTTP does not need to do any separate multiplexing when using QUIC - data sent over a QUIC stream always maps to a particular HTTP transaction or to the entire HTTP/3 connection context.

6.1. Bidirectional Streams

All client-initiated bidirectional streams are used for HTTP requests and responses. A bidirectional stream ensures that the response can be readily correlated with the request. These streams are referred to as request streams.

This means that the client's first request occurs on QUIC stream 0, with subsequent requests on stream 4, 8, and so on. In order to permit these streams to open, an HTTP/3 server SHOULD configure non-zero minimum values for the number of permitted streams and the initial stream flow control window. So as to not unnecessarily limit parallelism, at least 100 request streams SHOULD be permitted at a time.

HTTP/3 does not use server-initiated bidirectional streams, though an extension could define a use for these streams. Clients MUST treat receipt of a server-initiated bidirectional stream as a connection error of type H3_STREAM_CREATION_ERROR (Section 8) unless such an extension has been negotiated.

6.2. Unidirectional Streams

Unidirectional streams, in either direction, are used for a range of purposes. The purpose is indicated by a stream type, which is sent as a variable-length integer at the start of the stream. The format and structure of data that follows this integer is determined by the stream type.

```
Unidirectional Stream Header {
  Stream Type (i),
}
```

Figure 1: Unidirectional Stream Header

Two stream types are defined in this document: control streams (Section 6.2.1) and push streams (Section 6.2.2). [QPACK] defines two additional stream types. Other stream types can be defined by extensions to HTTP/3; see Section 9 for more details. Some stream types are reserved (Section 6.2.3).

The performance of HTTP/3 connections in the early phase of their lifetime is sensitive to the creation and exchange of data on unidirectional streams. Endpoints that excessively restrict the number of streams or the flow control window of these streams will increase the chance that the remote peer reaches the limit early and becomes blocked. In particular, implementations should consider that remote peers may wish to exercise reserved stream behavior (Section 6.2.3) with some of the unidirectional streams they are permitted to use. To avoid blocking, the transport parameters sent by both clients and servers MUST allow the peer to create at least one unidirectional stream for the HTTP control stream plus the number of unidirectional streams required by mandatory extensions (three being the minimum number required for the base HTTP/3 protocol and QPACK), and SHOULD provide at least 1,024 bytes of flow control credit to each stream.

Note that an endpoint is not required to grant additional credits to create more unidirectional streams if its peer consumes all the initial credits before creating the critical unidirectional streams. Endpoints SHOULD create the HTTP control stream as well as the unidirectional streams required by mandatory extensions (such as the QPACK encoder and decoder streams) first, and then create additional streams as allowed by their peer.

If the stream header indicates a stream type that is not supported by the recipient, the remainder of the stream cannot be consumed as the semantics are unknown. Recipients of unknown stream types MAY abort reading of the stream with an error code of

H3_STREAM_CREATION_ERROR or a reserved error code (Section 8.1), but MUST NOT consider such streams to be a connection error of any kind.

Implementations MAY send stream types before knowing whether the peer supports them. However, stream types that could modify the state or semantics of existing protocol components, including QPACK or other extensions, MUST NOT be sent until the peer is known to support them.

A sender can close or reset a unidirectional stream unless otherwise specified. A receiver MUST tolerate unidirectional streams being closed or reset prior to the reception of the unidirectional stream header.

6.2.1. Control Streams

A control stream is indicated by a stream type of 0x00. Data on this stream consists of HTTP/3 frames, as defined in Section 7.2.

Each side MUST initiate a single control stream at the beginning of the connection and send its SETTINGS frame as the first frame on this stream. If the first frame of the control stream is any other frame type, this MUST be treated as a connection error of type H3_MISSING_SETTINGS. Only one control stream per peer is permitted; receipt of a second stream claiming to be a control stream MUST be treated as a connection error of type H3_STREAM_CREATION_ERROR. The sender MUST NOT close the control stream, and the receiver MUST NOT request that the sender close the control stream. If either control stream is closed at any point, this MUST be treated as a connection error of type H3_CLOSED_CRITICAL_STREAM. Connection errors are described in Section 8.

Because the contents of the control stream are used to manage the behavior of other streams, endpoints SHOULD provide enough flow control credit to keep the peer's control stream from becoming blocked.

A pair of unidirectional streams is used rather than a single bidirectional stream. This allows either peer to send data as soon as it is able. Depending on whether 0-RTT is available on the QUIC connection, either client or server might be able to send stream data first.

6.2.2. Push Streams

Server push is an optional feature introduced in HTTP/2 that allows a server to initiate a response before a request has been made. See Section 4.4 for more details.

A push stream is indicated by a stream type of 0x01, followed by the Push ID of the promise that it fulfills, encoded as a variable-length integer. The remaining data on this stream consists of HTTP/3 frames, as defined in Section 7.2, and fulfills a promised server push by zero or more interim HTTP responses followed by a single final HTTP response, as defined in Section 4.1. Server push and Push IDs are described in Section 4.4.

Only servers can push; if a server receives a client-initiated push stream, this MUST be treated as a connection error of type H3_STREAM_CREATION_ERROR; see Section 8.

```
Push Stream Header {
  Stream Type (i) = 0x01,
  Push ID (i),
}
```

Figure 2: Push Stream Header

Each Push ID MUST only be used once in a push stream header. If a push stream header includes a Push ID that was used in another push stream header, the client MUST treat this as a connection error of type H3_ID_ERROR; see Section 8.

6.2.3. Reserved Stream Types

Stream types of the format 0x1f * N + 0x21 for non-negative integer values of N are reserved to exercise the requirement that unknown types be ignored. These streams have no semantics, and can be sent when application-layer padding is desired. They MAY also be sent on connections where no data is currently being transferred. Endpoints MUST NOT consider these streams to have any meaning upon receipt.

The payload and length of the stream are selected in any manner the sending implementation chooses. When sending a reserved stream type, the implementation MAY either terminate the stream cleanly or reset it. When resetting the stream, either the H3_NO_ERROR error code or a reserved error code (Section 8.1) SHOULD be used.

7. HTTP Framing Layer

HTTP frames are carried on QUIC streams, as described in Section 6. HTTP/3 defines three stream types: control stream, request stream, and push stream. This section describes HTTP/3 frame formats and their permitted stream types; see Table 1 for an overview. A comparison between HTTP/2 and HTTP/3 frames is provided in Appendix A.2.

Frame	Control Stream	Request Stream	Push Stream	Section
DATA	No	Yes	Yes	Section 7.2.1
HEADERS	No	Yes	Yes	Section 7.2.2
CANCEL_PUSH	Yes	No	No	Section 7.2.3
SETTINGS	Yes (1)	No	No	Section 7.2.4
PUSH_PROMISE	No	Yes	No	Section 7.2.5
GOAWAY	Yes	No	No	Section 7.2.6
MAX_PUSH_ID	Yes	No	No	Section 7.2.7
Reserved	Yes	Yes	Yes	Section 7.2.8

Table 1: HTTP/3 Frames and Stream Type Overview

The SETTINGS frame can only occur as the first frame of a Control stream; this is indicated in Table 1 with a (1). Specific guidance is provided in the relevant section.

Note that, unlike QUIC frames, HTTP/3 frames can span multiple packets.

7.1. Frame Layout

All frames have the following format:

```
HTTP/3 Frame Format {
  Type (i),
  Length (i),
  Frame Payload (..),
}
```

Figure 3: HTTP/3 Frame Format

A frame includes the following fields:

Type: A variable-length integer that identifies the frame type.

Length: A variable-length integer that describes the length in bytes of the Frame Payload.

Frame Payload: A payload, the semantics of which are determined by the Type field.

Each frame's payload MUST contain exactly the fields identified in its description. A frame payload that contains additional bytes after the identified fields or a frame payload that terminates before the end of the identified fields MUST be treated as a connection error of type H3_FRAME_ERROR; see Section 8. In particular, redundant length encodings MUST be verified to be self-consistent; see Section 10.8.

When a stream terminates cleanly, if the last frame on the stream was truncated, this MUST be treated as a connection error of type H3_FRAME_ERROR; see Section 8. Streams that terminate abruptly may be reset at any point in a frame.

7.2. Frame Definitions

7.2.1. DATA

DATA frames (type=0x0) convey arbitrary, variable-length sequences of bytes associated with HTTP request or response content.

DATA frames MUST be associated with an HTTP request or response. If a DATA frame is received on a control stream, the recipient MUST respond with a connection error of type H3_FRAME_UNEXPECTED; see Section 8.

```
DATA Frame {
  Type (i) = 0x0,
  Length (i),
  Data (..),
}
```

Figure 4: DATA Frame

7.2.2. HEADERS

The HEADERS frame (type=0x1) is used to carry an HTTP field section, encoded using QPACK. See [QPACK] for more details.

```
HEADERS Frame {
  Type (i) = 0x1,
  Length (i),
  Encoded Field Section (..),
}
```

Figure 5: HEADERS Frame

HEADERS frames can only be sent on request or push streams. If a HEADERS frame is received on a control stream, the recipient MUST respond with a connection error (Section 8) of type H3_FRAME_UNEXPECTED.

7.2.3. CANCEL_PUSH

The CANCEL_PUSH frame (type=0x3) is used to request cancellation of a server push prior to the push stream being received. The CANCEL_PUSH frame identifies a server push by Push ID (see Section 4.4), encoded as a variable-length integer.

When a client sends CANCEL_PUSH, it is indicating that it does not wish to receive the promised resource. The server SHOULD abort sending the resource, but the mechanism to do so depends on the state of the corresponding push stream. If the server has not yet created a push stream, it does not create one. If the push stream is open, the server SHOULD abruptly terminate that stream. If the push stream has already ended, the server MAY still abruptly terminate the stream or MAY take no action.

A server sends CANCEL_PUSH to indicate that it will not be fulfilling a promise which was previously sent. The client cannot expect the corresponding promise to be fulfilled, unless it has already received and processed the promised response. Regardless of whether a push stream has been opened, a server SHOULD send a CANCEL_PUSH frame when it determines that promise will not be fulfilled. If a stream has already been opened, the server can abort sending on the stream with an error code of H3_REQUEST_CANCELLED.

Sending a CANCEL_PUSH frame has no direct effect on the state of existing push streams. A client SHOULD NOT send a CANCEL_PUSH frame when it has already received a corresponding push stream. A push stream could arrive after a client has sent a CANCEL_PUSH frame, because a server might not have processed the CANCEL_PUSH. The client SHOULD abort reading the stream with an error code of H3_REQUEST_CANCELLED.

A CANCEL_PUSH frame is sent on the control stream. Receiving a CANCEL_PUSH frame on a stream other than the control stream MUST be treated as a connection error of type H3_FRAME_UNEXPECTED.

```
CANCEL_PUSH Frame {
  Type (i) = 0x3,
  Length (i),
  Push ID (i),
}
```

Figure 6: CANCEL_PUSH Frame

The CANCEL_PUSH frame carries a Push ID encoded as a variable-length integer. The Push ID identifies the server push that is being cancelled; see Section 4.4. If a CANCEL_PUSH frame is received that references a Push ID greater than currently allowed on the connection, this MUST be treated as a connection error of type H3_ID_ERROR.

If the client receives a CANCEL_PUSH frame, that frame might identify a Push ID that has not yet been mentioned by a PUSH_PROMISE frame due to reordering. If a server receives a CANCEL_PUSH frame for a Push ID that has not yet been mentioned by a PUSH_PROMISE frame, this MUST be treated as a connection error of type H3_ID_ERROR.

7.2.4. SETTINGS

The SETTINGS frame (type=0x4) conveys configuration parameters that affect how endpoints communicate, such as preferences and constraints on peer behavior. Individually, a SETTINGS parameter can also be referred to as a "setting"; the identifier and value of each setting parameter can be referred to as a "setting identifier" and a "setting value".

SETTINGS frames always apply to an entire HTTP/3 connection, never a single stream. A SETTINGS frame MUST be sent as the first frame of each control stream (see Section 6.2.1) by each peer, and MUST NOT be sent subsequently. If an endpoint receives a second SETTINGS frame on the control stream, the endpoint MUST respond with a connection error of type H3_FRAME_UNEXPECTED.

SETTINGS frames MUST NOT be sent on any stream other than the control stream. If an endpoint receives a SETTINGS frame on a different stream, the endpoint MUST respond with a connection error of type H3_FRAME_UNEXPECTED.

SETTINGS parameters are not negotiated; they describe characteristics of the sending peer that can be used by the receiving peer. However, a negotiation can be implied by the use of SETTINGS - each peer uses SETTINGS to advertise a set of supported values. The definition of the setting would describe how each peer combines the two sets to conclude which choice will be used. SETTINGS does not provide a mechanism to identify when the choice takes effect.

Different values for the same parameter can be advertised by each peer. For example, a client might be willing to consume a very large response field section, while servers are more cautious about request size.

The same setting identifier MUST NOT occur more than once in the SETTINGS frame. A receiver MAY treat the presence of duplicate setting identifiers as a connection error of type H3_SETTINGS_ERROR.

The payload of a SETTINGS frame consists of zero or more parameters. Each parameter consists of a setting identifier and a value, both encoded as QUIC variable-length integers.

```
Setting {
  Identifier (i),
  Value (i),
}

SETTINGS Frame {
  Type (i) = 0x4,
  Length (i),
  Setting (..) ...,
}
```

Figure 7: SETTINGS Frame

An implementation MUST ignore any parameter with an identifier it does not understand.

7.2.4.1. Defined SETTINGS Parameters

The following settings are defined in HTTP/3:

SETTINGS_MAX_FIELD_SECTION_SIZE (0x6): The default value is unlimited. See Section 4.1.1.3 for usage.

Setting identifiers of the format 0x1f * N + 0x21 for non-negative integer values of N are reserved to exercise the requirement that unknown identifiers be ignored. Such settings have no defined meaning. Endpoints SHOULD include at least one such setting in their SETTINGS frame. Endpoints MUST NOT consider such settings to have any meaning upon receipt.

Because the setting has no defined meaning, the value of the setting can be any value the implementation selects.

Setting identifiers which were defined in [HTTP2] where there is no corresponding HTTP/3 setting have also been reserved (Section 11.2.2). These reserved settings MUST NOT be sent, and their receipt MUST be treated as a connection error of type H3_SETTINGS_ERROR.

Additional settings can be defined by extensions to HTTP/3; see Section 9 for more details.

7.2.4.2. Initialization

An HTTP implementation MUST NOT send frames or requests that would be invalid based on its current understanding of the peer's settings.

All settings begin at an initial value. Each endpoint SHOULD use these initial values to send messages before the peer's SETTINGS frame has arrived, as packets carrying the settings can be lost or delayed. When the SETTINGS frame arrives, any settings are changed to their new values.

This removes the need to wait for the SETTINGS frame before sending messages. Endpoints MUST NOT require any data to be received from the peer prior to sending the SETTINGS frame; settings MUST be sent as soon as the transport is ready to send data.

For servers, the initial value of each client setting is the default value.

For clients using a 1-RTT QUIC connection, the initial value of each server setting is the default value. 1-RTT keys will always become available prior to the packet containing SETTINGS being processed by QUIC, even if the server sends SETTINGS immediately. Clients SHOULD NOT wait indefinitely for SETTINGS to arrive before sending requests, but SHOULD process received datagrams in order to increase the likelihood of processing SETTINGS before sending the first request.

When a 0-RTT QUIC connection is being used, the initial value of each server setting is the value used in the previous session. Clients SHOULD store the settings the server provided in the HTTP/3 connection where resumption information was provided, but MAY opt not to store settings in certain cases (e.g., if the session ticket is received before the SETTINGS frame). A client MUST comply with stored settings -- or default values, if no values are stored -- when attempting 0-RTT. Once a server has provided new settings, clients MUST comply with those values.

A server can remember the settings that it advertised, or store an integrity-protected copy of the values in the ticket and recover the information when accepting 0-RTT data. A server uses the HTTP/3 settings values in determining whether to accept 0-RTT data. If the server cannot determine that the settings remembered by a client are compatible with its current settings, it MUST NOT accept 0-RTT data. Remembered settings are compatible if a client complying with those settings would not violate the server's current settings.

A server MAY accept 0-RTT and subsequently provide different settings in its SETTINGS frame. If 0-RTT data is accepted by the server, its SETTINGS frame MUST NOT reduce any limits or alter any values that might be violated by the client with its 0-RTT data. The server MUST include all settings that differ from their default values. If a server accepts 0-RTT but then sends settings that are not compatible with the previously specified settings, this MUST be treated as a connection error of type H3_SETTINGS_ERROR. If a server accepts 0-RTT but then sends a SETTINGS frame that omits a setting value that the client understands (apart from reserved setting identifiers) that was previously specified to have a non-default value, this MUST be treated as a connection error of type H3_SETTINGS_ERROR.

7.2.5. PUSH_PROMISE

The PUSH_PROMISE frame (type=0x5) is used to carry a promised request header section from server to client on a request stream, as in HTTP/2.

```
PUSH_PROMISE Frame {
  Type (i) = 0x5,
  Length (i),
  Push ID (i),
  Encoded Field Section (..),
}
```

Figure 8: PUSH_PROMISE Frame

The payload consists of:

Push ID: A variable-length integer that identifies the server push operation. A Push ID is used in push stream headers (Section 4.4) and CANCEL_PUSH frames (Section 7.2.3).

Encoded Field Section: QPACK-encoded request header fields for the promised response. See [QPACK] for more details.

A server MUST NOT use a Push ID that is larger than the client has provided in a MAX_PUSH_ID frame (Section 7.2.7). A client MUST treat receipt of a PUSH_PROMISE frame that contains a larger Push ID than the client has advertised as a connection error of H3_ID_ERROR.

A server MAY use the same Push ID in multiple PUSH_PROMISE frames. If so, the decompressed request header sets MUST contain the same fields in the same order, and both the name and the value in each field MUST be exact matches. Clients SHOULD compare the request header sections for resources promised multiple times. If a client receives a Push ID that has already been promised and detects a mismatch, it MUST respond with a connection error of type H3_GENERAL_PROTOCOL_ERROR. If the decompressed field sections match exactly, the client SHOULD associate the pushed content with each stream on which a PUSH_PROMISE frame was received.

Allowing duplicate references to the same Push ID is primarily to reduce duplication caused by concurrent requests. A server SHOULD avoid reusing a Push ID over a long period. Clients are likely to consume server push responses and not retain them for reuse over time. Clients that see a PUSH_PROMISE frame that uses a Push ID that they have already consumed and discarded are forced to ignore the promise.

If a PUSH_PROMISE frame is received on the control stream, the client MUST respond with a connection error of type H3_FRAME_UNEXPECTED; see Section 8.

A client MUST NOT send a PUSH_PROMISE frame. A server MUST treat the receipt of a PUSH_PROMISE frame as a connection error of type H3_FRAME_UNEXPECTED; see Section 8.

See Section 4.4 for a description of the overall server push mechanism.

7.2.6. GOAWAY

The GOAWAY frame (type=0x7) is used to initiate graceful shutdown of an HTTP/3 connection by either endpoint. GOAWAY allows an endpoint to stop accepting new requests or pushes while still finishing processing of previously received requests and pushes. This enables administrative actions, like server maintenance. GOAWAY by itself does not close a connection.

```
GOAWAY Frame {
  Type (i) = 0x7,
  Length (i),
  Stream ID/Push ID (..),
}
```

Figure 9: GOAWAY Frame

The GOAWAY frame is always sent on the control stream. In the server to client direction, it carries a QUIC Stream ID for a client-initiated bidirectional stream encoded as a variable-length integer. A client MUST treat receipt of a GOAWAY frame containing a Stream ID of any other type as a connection error of type H3_ID_ERROR.

In the client to server direction, the GOAWAY frame carries a Push ID encoded as a variable-length integer.

The GOAWAY frame applies to the entire connection, not a specific stream. A client MUST treat a GOAWAY frame on a stream other than the control stream as a connection error of type H3_FRAME_UNEXPECTED; see Section 8.

See Section 5.2 for more information on the use of the GOAWAY frame.

7.2.7. MAX_PUSH_ID

The MAX_PUSH_ID frame (type=0xd) is used by clients to control the number of server pushes that the server can initiate. This sets the maximum value for a Push ID that the server can use in PUSH_PROMISE and CANCEL_PUSH frames. Consequently, this also limits the number of push streams that the server can initiate in addition to the limit maintained by the QUIC transport.

The MAX_PUSH_ID frame is always sent on the control stream. Receipt of a MAX_PUSH_ID frame on any other stream MUST be treated as a connection error of type H3_FRAME_UNEXPECTED.

A server MUST NOT send a MAX_PUSH_ID frame. A client MUST treat the receipt of a MAX_PUSH_ID frame as a connection error of type H3_FRAME_UNEXPECTED.

The maximum Push ID is unset when an HTTP/3 connection is created, meaning that a server cannot push until it receives a MAX_PUSH_ID frame. A client that wishes to manage the number of promised server pushes can increase the maximum Push ID by sending MAX_PUSH_ID frames as the server fulfills or cancels server pushes.

```
MAX_PUSH_ID Frame {
  Type (i) = 0xd,
  Length (i),
  Push ID (i),
}
```

Figure 10: MAX_PUSH_ID Frame

The MAX_PUSH_ID frame carries a single variable-length integer that identifies the maximum value for a Push ID that the server can use; see Section 4.4. A MAX_PUSH_ID frame cannot reduce the maximum Push ID; receipt of a MAX_PUSH_ID frame that contains a smaller value than previously received MUST be treated as a connection error of type H3_ID_ERROR.

7.2.8. Reserved Frame Types

Frame types of the format 0x1f * N + 0x21 for non-negative integer values of N are reserved to exercise the requirement that unknown types be ignored (Section 9). These frames have no semantics, and MAY be sent on any stream where frames are allowed to be sent. This enables their use for application-layer padding. Endpoints MUST NOT consider these frames to have any meaning upon receipt.

The payload and length of the frames are selected in any manner the implementation chooses.

Frame types that were used in HTTP/2 where there is no corresponding HTTP/3 frame have also been reserved (Section 11.2.1). These frame types MUST NOT be sent, and their receipt MUST be treated as a connection error of type H3_FRAME_UNEXPECTED.

8. Error Handling

When a stream cannot be completed successfully, QUIC allows the application to abruptly terminate (reset) that stream and communicate a reason; see Section 2.4 of [QUIC-TRANSPORT]. This is referred to as a "stream error." An HTTP/3 implementation can decide to close a QUIC stream and communicate the type of error. Wire encodings of error codes are defined in Section 8.1. Stream errors are distinct from HTTP status codes which indicate error conditions. Stream errors indicate that the sender did not transfer or consume the full request or response, while HTTP status codes indicate the result of a request that was successfully received.

If an entire connection needs to be terminated, QUIC similarly provides mechanisms to communicate a reason; see Section 5.3 of [QUIC-TRANSPORT]. This is referred to as a "connection error." Similar to stream errors, an HTTP/3 implementation can terminate a QUIC connection and communicate the reason using an error code from Section 8.1.

Although the reasons for closing streams and connections are called "errors," these actions do not necessarily indicate a problem with the connection or either implementation. For example, a stream can be reset if the requested resource is no longer needed.

An endpoint MAY choose to treat a stream error as a connection error under certain circumstances, closing the entire connection in response to a condition on a single stream. Implementations need to consider the impact on outstanding requests before making this choice.

Because new error codes can be defined without negotiation (see Section 9), use of an error code in an unexpected context or receipt of an unknown error code MUST be treated as equivalent to H3_NO_ERROR. However, closing a stream can have other effects regardless of the error code; for example, see Section 4.1.

8.1. HTTP/3 Error Codes

The following error codes are defined for use when abruptly terminating streams, aborting reading of streams, or immediately closing HTTP/3 connections.

H3_NO_ERROR (0x100): No error. This is used when the connection or stream needs to be closed, but there is no error to signal.

H3_GENERAL_PROTOCOL_ERROR (0x101): Peer violated protocol requirements in a way that does not match a more specific error code, or endpoint declines to use the more specific error code.

H3_INTERNAL_ERROR (0x102): An internal error has occurred in the HTTP stack.

H3_STREAM_CREATION_ERROR (0x103): The endpoint detected that its peer created a stream that it will not accept.

H3_CLOSED_CRITICAL_STREAM (0x104): A stream required by the HTTP/3 connection was closed or reset.

H3_FRAME_UNEXPECTED (0x105): A frame was received that was not permitted in the current state or on the current stream.

H3_FRAME_ERROR (0x106): A frame that fails to satisfy layout requirements or with an invalid size was received.

H3_EXCESSIVE_LOAD (0x107): The endpoint detected that its peer is exhibiting a behavior that might be generating excessive load.

H3_ID_ERROR (0x108): A Stream ID or Push ID was used incorrectly, such as exceeding a limit, reducing a limit, or being reused.

H3_SETTINGS_ERROR (0x109): An endpoint detected an error in the payload of a SETTINGS frame.

H3_MISSING_SETTINGS (0x10a): No SETTINGS frame was received at the beginning of the control stream.

H3_REQUEST_REJECTED (0x10b): A server rejected a request without performing any application processing.

H3_REQUEST_CANCELLED (0x10c): The request or its response (including pushed response) is cancelled.

H3_REQUEST_INCOMPLETE (0x10d): The client's stream terminated without containing a fully-formed request.

H3_MESSAGE_ERROR (0x10e): An HTTP message was malformed and cannot be processed.

H3_CONNECT_ERROR (0x10f): The TCP connection established in response to a CONNECT request was reset or abnormally closed.

H3_VERSION_FALLBACK (0x110):
The requested operation cannot be served over HTTP/3. The peer should retry over HTTP/1.1.

Error codes of the format 0x1f * N + 0x21 for non-negative integer values of N are reserved to exercise the requirement that unknown error codes be treated as equivalent to H3_NO_ERROR (Section 9). Implementations SHOULD select an error code from this space with some probability when they would have sent H3_NO_ERROR.

9. Extensions to HTTP/3

HTTP/3 permits extension of the protocol. Within the limitations described in this section, protocol extensions can be used to provide additional services or alter any aspect of the protocol. Extensions are effective only within the scope of a single HTTP/3 connection.

This applies to the protocol elements defined in this document. This does not affect the existing options for extending HTTP, such as defining new methods, status codes, or fields.

Extensions are permitted to use new frame types (Section 7.2), new settings (Section 7.2.4.1), new error codes (Section 8), or new unidirectional stream types (Section 6.2). Registries are established for managing these extension points: frame types (Section 11.2.1), settings (Section 11.2.2), error codes (Section 11.2.3), and stream types (Section 11.2.4).

Implementations MUST ignore unknown or unsupported values in all extensible protocol elements. Implementations MUST discard frames and abort reading on unidirectional streams that have unknown or unsupported types. This means that any of these extension points can be safely used by extensions without prior arrangement or negotiation. However, where a known frame type is required to be in a specific location, such as the SETTINGS frame as the first frame of the control stream (see Section 6.2.1), an unknown frame type does not satisfy that requirement and SHOULD be treated as an error.

Extensions that could change the semantics of existing protocol components MUST be negotiated before being used. For example, an extension that changes the layout of the HEADERS frame cannot be used until the peer has given a positive signal that this is acceptable. Coordinating when such a revised layout comes into effect could prove complex. As such, allocating new identifiers for new definitions of existing protocol elements is likely to be more effective.

This document does not mandate a specific method for negotiating the use of an extension but notes that a setting (Section 7.2.4.1) could be used for that purpose. If both peers set a value that indicates willingness to use the extension, then the extension can be used. If a setting is used for extension negotiation, the default value MUST be defined in such a fashion that the extension is disabled if the setting is omitted.

10. Security Considerations

The security considerations of HTTP/3 should be comparable to those of HTTP/2 with TLS. However, many of the considerations from Section 10 of [HTTP2] apply to [QUIC-TRANSPORT] and are discussed in that document.

10.1. Server Authority

HTTP/3 relies on the HTTP definition of authority. The security considerations of establishing authority are discussed in Section 17.1 of [SEMANTICS].

10.2. Cross-Protocol Attacks

The use of ALPN in the TLS and QUIC handshakes establishes the target application protocol before application-layer bytes are processed. This ensures that endpoints have strong assurances that peers are using the same protocol.

This does not guarantee protection from all cross-protocol attacks. Section 21.5 of [QUIC-TRANSPORT] describes some ways in which the plaintext of QUIC packets can be used to perform request forgery against endpoints that don't use authenticated transports.

10.3. Intermediary Encapsulation Attacks

The HTTP/3 field encoding allows the expression of names that are not valid field names in the syntax used by HTTP (Section 5.1 of [SEMANTICS]). Requests or responses containing invalid field names MUST be treated as malformed (Section 4.1.3). An intermediary therefore cannot translate an HTTP/3 request or response containing an invalid field name into an HTTP/1.1 message.

Similarly, HTTP/3 can transport field values that are not valid. While most values that can be encoded will not alter field parsing, carriage return (CR, ASCII 0xd), line feed (LF, ASCII 0xa), and the zero character (NUL, ASCII 0x0) might be exploited by an attacker if they are translated verbatim. Any request or response that contains a character not permitted in a field value MUST be treated as malformed (Section 4.1.3). Valid characters are defined by the "field-content" ABNF rule in Section 5.5 of [SEMANTICS].

10.4. Cacheability of Pushed Responses

Pushed responses do not have an explicit request from the client; the request is provided by the server in the PUSH_PROMISE frame.

Caching responses that are pushed is possible based on the guidance provided by the origin server in the Cache-Control header field. However, this can cause issues if a single server hosts more than one tenant. For example, a server might offer multiple users each a small portion of its URI space.

Where multiple tenants share space on the same server, that server MUST ensure that tenants are not able to push representations of resources that they do not have authority over. Failure to enforce this would allow a tenant to provide a representation that would be served out of cache, overriding the actual representation that the authoritative tenant provides.

Clients are required to reject pushed responses for which an origin server is not authoritative; see Section 4.4.

10.5. Denial-of-Service Considerations

An HTTP/3 connection can demand a greater commitment of resources to operate than an HTTP/1.1 or HTTP/2 connection. The use of field compression and flow control depend on a commitment of resources for storing a greater amount of state. Settings for these features ensure that memory commitments for these features are strictly bounded.

The number of PUSH_PROMISE frames is constrained in a similar fashion. A client that accepts server push SHOULD limit the number of Push IDs it issues at a time.

Processing capacity cannot be guarded as effectively as state capacity.

The ability to send undefined protocol elements that the peer is required to ignore can be abused to cause a peer to expend additional processing time. This might be done by setting multiple undefined SETTINGS parameters, unknown frame types, or unknown stream types. Note, however, that some uses are entirely legitimate, such as optional-to-understand extensions and padding to increase resistance to traffic analysis.

Compression of field sections also offers some opportunities to waste processing resources; see Section 7 of [QPACK] for more details on potential abuses.

All these features -- i.e., server push, unknown protocol elements, field compression -- have legitimate uses. These features become a burden only when they are used unnecessarily or to excess.

An endpoint that does not monitor such behavior exposes itself to a risk of denial-of-service attack. Implementations SHOULD track the use of these features and set limits on their use. An endpoint MAY treat activity that is suspicious as a connection error of type H3_EXCESSIVE_LOAD (Section 8), but false positives will result in disrupting valid connections and requests.

10.5.1. Limits on Field Section Size

A large field section (Section 4.1) can cause an implementation to commit a large amount of state. Header fields that are critical for routing can appear toward the end of a header section, which prevents streaming of the header section to its ultimate destination. This ordering and other reasons, such as ensuring cache correctness, mean that an endpoint likely needs to buffer the entire header section. Since there is no hard limit to the size of a field section, some endpoints could be forced to commit a large amount of available memory for header fields.

An endpoint can use the SETTINGS_MAX_FIELD_SECTION_SIZE (Section 4.1.1.3) setting to advise peers of limits that might apply on the size of field sections. This setting is only advisory, so endpoints MAY choose to send field sections that exceed this limit and risk having the request or response being treated as malformed. This setting is specific to an HTTP/3 connection, so any request or response could encounter a hop with a lower, unknown limit. An intermediary can attempt to avoid this problem by passing on values presented by different peers, but they are not obligated to do so.

A server that receives a larger field section than it is willing to handle can send an HTTP 431 (Request Header Fields Too Large) status code ([RFC6585]). A client can discard responses that it cannot process.

10.5.2. CONNECT Issues

The CONNECT method can be used to create disproportionate load on a proxy, since stream creation is relatively inexpensive when compared to the creation and maintenance of a TCP connection. Therefore, a proxy that supports CONNECT might be more conservative in the number of simultaneous requests it accepts.

A proxy might also maintain some resources for a TCP connection beyond the closing of the stream that carries the CONNECT request, since the outgoing TCP connection remains in the TIME_WAIT state. To account for this, a proxy might delay increasing the QUIC stream limits for some time after a TCP connection terminates.

10.6. Use of Compression

Compression can allow an attacker to recover secret data when it is compressed in the same context as data under attacker control. HTTP/3 enables compression of fields (Section 4.1.1); the following concerns also apply to the use of HTTP compressed content-codings; see Section 8.4.1 of [SEMANTICS].

There are demonstrable attacks on compression that exploit the characteristics of the web (e.g., [BREACH]). The attacker induces multiple requests containing varying plaintext, observing the length of the resulting ciphertext in each, which reveals a shorter length when a guess about the secret is correct.

Implementations communicating on a secure channel MUST NOT compress content that includes both confidential and attacker-controlled data unless separate compression contexts are used for each source of data. Compression MUST NOT be used if the source of data cannot be reliably determined.

Further considerations regarding the compression of field sections are described in [QPACK].

10.7. Padding and Traffic Analysis

Padding can be used to obscure the exact size of frame content and is provided to mitigate specific attacks within HTTP, for example, attacks where compressed content includes both attacker-controlled plaintext and secret data (e.g., [BREACH]).

Where HTTP/2 employs PADDING frames and Padding fields in other frames to make a connection more resistant to traffic analysis, HTTP/3 can either rely on transport-layer padding or employ the reserved frame and stream types discussed in Section 7.2.8 and Section 6.2.3. These methods of padding produce different results in terms of the granularity of padding, how padding is arranged in relation to the information that is being protected, whether padding is applied in the case of packet loss, and how an implementation might control padding.

Reserved stream types can be used to give the appearance of sending traffic even when the connection is idle. Because HTTP traffic often occurs in bursts, apparent traffic can be used to obscure the timing or duration of such bursts, even to the point of appearing to send a constant stream of data. However, as such traffic is still flow controlled by the receiver, a failure to promptly drain such streams and provide additional flow control credit can limit the sender's ability to send real traffic.

To mitigate attacks that rely on compression, disabling or limiting compression might be preferable to padding as a countermeasure.

Use of padding can result in less protection than might seem immediately obvious. Redundant padding could even be counterproductive. At best, padding only makes it more difficult for an attacker to infer length information by increasing the number of frames an attacker has to observe. Incorrectly implemented padding schemes can be easily defeated. In particular, randomized padding with a predictable distribution provides very little protection; similarly, padding payloads to a fixed size exposes information as payload sizes cross the fixed-sized boundary, which could be possible if an attacker can control plaintext.

10.8. Frame Parsing

Several protocol elements contain nested length elements, typically in the form of frames with an explicit length containing variable-length integers. This could pose a security risk to an incautious implementer. An implementation MUST ensure that the length of a frame exactly matches the length of the fields it contains.

10.9. Early Data

The use of 0-RTT with HTTP/3 creates an exposure to replay attack. The anti-replay mitigations in [HTTP-REPLAY] MUST be applied when using HTTP/3 with 0-RTT. When applying [HTTP-REPLAY] to HTTP/3, references to the TLS layer refer to the handshake performed within QUIC, while all references to application data refer to the contents of streams.

10.10. Migration

Certain HTTP implementations use the client address for logging or access-control purposes. Since a QUIC client's address might change during a connection (and future versions might support simultaneous use of multiple addresses), such implementations will need to either actively retrieve the client's current address or addresses when they are relevant or explicitly accept that the original address might change.

10.11. Privacy Considerations

Several characteristics of HTTP/3 provide an observer an opportunity to correlate actions of a single client or server over time. These include the value of settings, the timing of reactions to stimulus, and the handling of any features that are controlled by settings.

As far as these create observable differences in behavior, they could be used as a basis for fingerprinting a specific client.

HTTP/3's preference for using a single QUIC connection allows correlation of a user's activity on a site. Reusing connections for different origins allows for correlation of activity across those origins.

Several features of QUIC solicit immediate responses and can be used by an endpoint to measure latency to their peer; this might have privacy implications in certain scenarios.

11. IANA Considerations

This document registers a new ALPN protocol ID (Section 11.1) and creates new registries that manage the assignment of codepoints in HTTP/3.

11.1. Registration of HTTP/3 Identification String

This document creates a new registration for the identification of HTTP/3 in the "Application Layer Protocol Negotiation (ALPN) Protocol IDs" registry established in [RFC7301].

The "h3" string identifies HTTP/3:

Protocol: HTTP/3

Identification Sequence: 0x68 0x33 ("h3")

Specification: This document

11.2. New Registries

New registries created in this document operate under the QUIC registration policy documented in Section 22.1 of [QUIC-TRANSPORT]. These registries all include the common set of fields listed in Section 22.1.1 of [QUIC-TRANSPORT]. These registries [SHALL be/are] collected under a "Hypertext Transfer Protocol version 3 (HTTP/3) Parameters" heading.

The initial allocations in these registries created in this document are all assigned permanent status and list a change controller of the IETF and a contact of the HTTP working group (ietf-http-wg@w3.org).

11.2.1. Frame Types

This document establishes a registry for HTTP/3 frame type codes. The "HTTP/3 Frame Type" registry governs a 62-bit space. This registry follows the QUIC registry policy; see Section 11.2. Permanent registrations in this registry are assigned using the Specification Required policy ([RFC8126]), except for values between 0x00 and 0x3f (in hexadecimal; inclusive), which are assigned using Standards Action or IESG Approval as defined in Sections 4.9 and 4.10 of [RFC8126].

While this registry is separate from the "HTTP/2 Frame Type" registry defined in [HTTP2], it is preferable that the assignments parallel each other where the code spaces overlap. If an entry is present in only one registry, every effort SHOULD be made to avoid assigning the corresponding value to an unrelated operation. Expert reviewers MAY reject unrelated registrations which would conflict with the same value in the corresponding registry.

In addition to common fields as described in Section 11.2, permanent registrations in this registry MUST include the following field:

Frame Type: A name or label for the frame type.

Specifications of frame types MUST include a description of the frame layout and its semantics, including any parts of the frame that are conditionally present.

The entries in Table 2 are registered by this document.

Frame Type	Value	Specification
DATA	0x0	Section 7.2.1
HEADERS	0x1	Section 7.2.2
Reserved	0x2	N/A
CANCEL_PUSH	0x3	Section 7.2.3
SETTINGS	0x4	Section 7.2.4
PUSH_PROMISE	0x5	Section 7.2.5
Reserved	0x6	N/A
GOAWAY	0x7	Section 7.2.6
Reserved	0x8	N/A
Reserved	0x9	N/A
MAX_PUSH_ID	0xd	Section 7.2.7

Table 2: Initial HTTP/3 Frame Types

Each code of the format 0x1f * N + 0x21 for non-negative integer values of N (that is, 0x21, 0x40, ..., through 0x3ffffffffffffffe) MUST NOT be assigned by IANA and MUST NOT appear in the listing of assigned values.

11.2.2. Settings Parameters

This document establishes a registry for HTTP/3 settings. The "HTTP/3 Settings" registry governs a 62-bit space. This registry follows the QUIC registry policy; see Section 11.2. Permanent registrations in this registry are assigned using the Specification Required policy ([RFC8126]), except for values between 0x00 and 0x3f (in hexadecimal; inclusive), which are assigned using Standards Action or IESG Approval as defined in Sections 4.9 and 4.10 of [RFC8126].

While this registry is separate from the "HTTP/2 Settings" registry defined in [HTTP2], it is preferable that the assignments parallel each other. If an entry is present in only one registry, every effort SHOULD be made to avoid assigning the corresponding value to an unrelated operation. Expert reviewers MAY reject unrelated registrations which would conflict with the same value in the corresponding registry.

In addition to common fields as described in Section 11.2, permanent registrations in this registry MUST include the following fields:

Setting Name: A symbolic name for the setting. Specifying a setting name is optional.

Default: The value of the setting unless otherwise indicated. A default SHOULD be the most restrictive possible value.

The entries in Table 3 are registered by this document.

Setting Name	Value	Specification	Default
Reserved	0x0	N/A	N/A
Reserved	0x2	N/A	N/A
Reserved	0x3	N/A	N/A
Reserved	0x4	N/A	N/A
Reserved	0x5	N/A	N/A
MAX_FIELD_SECTION_SIZE	0x6	Section 7.2.4.1	Unlimited

Table 3: Initial HTTP/3 Settings

Each code of the format 0x1f * N + 0x21 for non-negative integer values of N (that is, 0x21, 0x40, ..., through 0x3ffffffffffffffe) MUST NOT be assigned by IANA and MUST NOT appear in the listing of assigned values.

11.2.3. Error Codes

This document establishes a registry for HTTP/3 error codes. The "HTTP/3 Error Code" registry manages a 62-bit space. This registry follows the QUIC registry policy; see Section 11.2. Permanent registrations in this registry are assigned using the Specification Required policy ([RFC8126]), except for values between 0x00 and 0x3f (in hexadecimal; inclusive), which are assigned using Standards Action or IESG Approval as defined in Sections 4.9 and 4.10 of [RFC8126].

Registrations for error codes are required to include a description of the error code. An expert reviewer is advised to examine new registrations for possible duplication with existing error codes. Use of existing registrations is to be encouraged, but not mandated. Use of values that are registered in the "HTTP/2 Error Code" registry is discouraged, and expert reviewers MAY reject such registrations.

In addition to common fields as described in Section 11.2, this registry includes two additional fields. Permanent registrations in this registry MUST include the following field:

Name: A name for the error code.

Description: A brief description of the error code semantics.

The entries in Table 4 are registered by this document. These error codes were selected from the range that operates on a Specification Required policy to avoid collisions with HTTP/2 error codes.

Name	Value	Description	Specification
H3_NO_ERROR	0x100	No error	Section 8.1
H3_GENERAL_PROTOCOL_ERROR	0x101	General protocol error	Section 8.1
H3_INTERNAL_ERROR	0x102	Internal error	Section 8.1
H3_STREAM_CREATION_ERROR	0x103	Stream creation error	Section 8.1
H3_CLOSED_CRITICAL_STREAM	0x104	Critical stream was closed	Section 8.1
H3_FRAME_UNEXPECTED	0x105	Frame not permitted in the current state	Section 8.1
H3_FRAME_ERROR	0x106	Frame violated layout or size rules	Section 8.1
H3_EXCESSIVE_LOAD	0x107	Peer generating excessive load	Section 8.1
H3_ID_ERROR	0x108	An identifier was used incorrectly	Section 8.1
H3_SETTINGS_ERROR	0x109	SETTINGS frame contained invalid values	Section 8.1
H3_MISSING_SETTINGS	0x10a	No SETTINGS frame received	Section 8.1

Name	Value	Description	Specification
H3_REQUEST_REJECTED	0x10b	Request not processed	Section 8.1
H3_REQUEST_CANCELLED	0x10c	Data no longer needed	Section 8.1
H3_REQUEST_INCOMPLETE	0x10d	Stream terminated early	Section 8.1
H3_MESSAGE_ERROR	0x10e	Malformed message	Section 8.1
H3_CONNECT_ERROR	0x10f	TCP reset or error on CONNECT request	Section 8.1
H3_VERSION_FALLBACK	0x110	Retry over HTTP/1.1	Section 8.1

Table 4: Initial HTTP/3 Error Codes

Each code of the format 0x1f * N + 0x21 for non-negative integer values of N (that is, 0x21, 0x40, ..., through 0x3ffffffffffffffe) MUST NOT be assigned by IANA and MUST NOT appear in the listing of assigned values.

11.2.4. Stream Types

This document establishes a registry for HTTP/3 unidirectional stream types. The "HTTP/3 Stream Type" registry governs a 62-bit space. This registry follows the QUIC registry policy; see Section 11.2. Permanent registrations in this registry are assigned using the Specification Required policy ([RFC8126]), except for values between 0x00 and 0x3f (in hexadecimal; inclusive), which are assigned using Standards Action or IESG Approval as defined in Sections 4.9 and 4.10 of [RFC8126].

In addition to common fields as described in Section 11.2, permanent registrations in this registry MUST include the following fields:

Stream Type: A name or label for the stream type.

Sender: Which endpoint on an HTTP/3 connection may initiate a stream of this type. Values are "Client", "Server", or "Both".

Specifications for permanent registrations MUST include a description of the stream type, including the layout and semantics of the stream contents.

The entries in the following table are registered by this document.

Stream Type	Value	Specification	Sender
Control Stream	0x00	Section 6.2.1	Both
Push Stream	0x01	Section 4.4	Server

Table 5

Each code of the format 0x1f * N + 0x21 for non-negative integer values of N (that is, 0x21, 0x40, ..., through 0x3ffffffffffffffe) MUST NOT be assigned by IANA and MUST NOT appear in the listing of assigned values.

12. References

12.1. Normative References

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

[CACHING] Fielding, R., Nottingham, M., and J. Reschke, "HTTP Caching", Work in Progress, Internet-Draft, draft-ietf-httpbis-cache-14, 12 January 2021, <http://www.ietf.org/internet-drafts/draft-ietf-httpbis-cache-14.txt>.

[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/info/rfc8470>.

[QPACK] Krasic, C., Bishop, M., and A. Frindell, Ed., "QPACK: Header Compression for HTTP over QUIC", Work in Progress, Internet-Draft, draft-ietf-quic-qpack-21, 2 February 2021, <https://tools.ietf.org/html/draft-ietf-quic-qpack-21>.

[QUIC-TRANSPORT] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", Work in Progress, Internet-Draft, draft-ietf-quic-transport-34, 2 February 2021, <https://tools.ietf.org/html/draft-ietf-quic-transport-34>.

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <https://www.rfc-editor.org/info/rfc793>.

[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>.

[RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI

10.17487/RFC6066, January 2011, <https://www.rfc-editor.org/info/rfc6066>.

[RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <https://www.rfc-editor.org/info/rfc6265>.

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

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

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

[SEMANTICS] Fielding, R., Nottingham, M., and J. Reschke, "HTTP Semantics", Work in Progress, Internet-Draft, draft-ietf-httpbis-semantics-14, 12 January 2021, <http://www.ietf.org/internet-drafts/draft-ietf-httpbis-semantics-14.txt>.

[URI] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

12.2. Informative References

[BREACH] Gluck, Y., Harris, N., and A. Prado, "BREACH: Reviving the CRIME Attack", July 2013, <http://breachattack.com/resources/BREACH%20-%20SSL,%20gone%20in%2030%20seconds.pdf>.

[DNS-TERMS] Hoffman, P., Sullivan, A., and K. Fujiwara, "DNS Terminology", BCP 219, RFC 8499, DOI 10.17487/RFC8499, January 2019, <https://www.rfc-editor.org/info/rfc8499>.

[HPACK] Peon, R. and H. Ruellan, "HPACK: Header Compression for HTTP/2", RFC 7541, DOI 10.17487/RFC7541, May 2015, <https://www.rfc-editor.org/info/rfc7541>.

[HTTP11] Fielding, R., Nottingham, M., and J. Reschke, "HTTP/1.1", Work in Progress, Internet-Draft, draft-ietf-httpbis-messaging-14, 12 January 2021, <http://www.ietf.org/internet-drafts/draft-ietf-httpbis-messaging-14.txt>.

[HTTP2] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <https://www.rfc-editor.org/info/rfc7540>.

[RFC6585] Nottingham, M. and R. Fielding, "Additional HTTP Status Codes", RFC 6585, DOI 10.17487/RFC6585, April 2012, <https://www.rfc-editor.org/info/rfc6585>.

[RFC8164] Nottingham, M. and M. Thomson, "Opportunistic Security for HTTP/2", RFC 8164, DOI 10.17487/RFC8164, May 2017, <https://www.rfc-editor.org/info/rfc8164>.

[TFO] 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/info/rfc7413>.

[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/info/rfc8446>.

Appendix A. Considerations for Transitioning from HTTP/2

HTTP/3 is strongly informed by HTTP/2, and bears many similarities. This section describes the approach taken to design HTTP/3, points out important differences from HTTP/2, and describes how to map HTTP/2 extensions into HTTP/3.

HTTP/3 begins from the premise that similarity to HTTP/2 is preferable, but not a hard requirement. HTTP/3 departs from HTTP/2 where QUIC differs from TCP, either to take advantage of QUIC features (like streams) or to accommodate important shortcomings (such as a lack of total ordering). These differences make HTTP/3 similar to HTTP/2 in key aspects, such as the relationship of requests and responses to streams. However, the details of the HTTP/3 design are substantially different from HTTP/2.

Some important departures are noted in this section.

A.1. Streams

HTTP/3 permits use of a larger number of streams (2^{62}-1) than HTTP/2. The same considerations about exhaustion of stream identifier space apply, though the space is significantly larger such that it is likely that other limits in QUIC are reached first, such as the limit on the connection flow control window.

In contrast to HTTP/2, stream concurrency in HTTP/3 is managed by QUIC. QUIC considers a stream closed when all data has been received and sent data has been acknowledged by the peer. HTTP/2 considers a stream closed when the frame containing the END_STREAM bit has been committed to the transport. As a result, the stream for an equivalent exchange could remain "active" for a longer period of time. HTTP/3 servers might choose to permit a larger number of concurrent client-initiated bidirectional streams to achieve equivalent concurrency to HTTP/2, depending on the expected usage patterns.

In HTTP/2, only request and response bodies (the frame payload of DATA frames) are subject to flow control. All HTTP/3 frames are sent on QUIC streams, so all frames on all streams are flow-controlled in HTTP/3.

Due to the presence of other unidirectional stream types, HTTP/3 does not rely exclusively on the number of concurrent unidirectional streams to control the number of concurrent in-flight pushes. Instead, HTTP/3 clients use the MAX_PUSH_ID frame to control the number of pushes received from an HTTP/3 server.

A.2. HTTP Frame Types

Many framing concepts from HTTP/2 can be elided on QUIC, because the transport deals with them. Because frames are already on a stream, they can omit the stream number. Because frames do not block multiplexing (QUIC's multiplexing occurs below this layer), the support for variable-maximum-length packets can be removed. Because stream termination is handled by QUIC, an END_STREAM flag is not required. This permits the removal of the Flags field from the generic frame layout.

Frame payloads are largely drawn from [HTTP2]. However, QUIC includes many features (e.g., flow control) that are also present in HTTP/2. In these cases, the HTTP mapping does not re-implement them. As a result, several HTTP/2 frame types are not required in HTTP/3. Where an HTTP/2-defined frame is no longer used, the frame ID has been reserved in order to maximize portability between HTTP/2 and HTTP/3 implementations. However, even frame types that appear in both mappings do not have identical semantics.

Many of the differences arise from the fact that HTTP/2 provides an absolute ordering between frames across all streams, while QUIC provides this guarantee on each stream only. As a result, if a frame type makes assumptions that frames from different streams will still be received in the order sent, HTTP/3 will break them.

Some examples of feature adaptations are described below, as well as general guidance to extension frame implementors converting an HTTP/2 extension to HTTP/3.

A.2.1. Prioritization Differences

HTTP/2 specifies priority assignments in PRIORITY frames and (optionally) in HEADERS frames. HTTP/3 does not provide a means of signaling priority.

Note that while there is no explicit signaling for priority, this does not mean that prioritization is not important for achieving good performance.

A.2.2. Field Compression Differences

HPACK was designed with the assumption of in-order delivery. A sequence of encoded field sections must arrive (and be decoded) at an endpoint in the same order in which they were encoded. This ensures that the dynamic state at the two endpoints remains in sync.

Because this total ordering is not provided by QUIC, HTTP/3 uses a modified version of HPACK, called QPACK. QPACK uses a single unidirectional stream to make all modifications to the dynamic table, ensuring a total order of updates. All frames that contain encoded fields merely reference the table state at a given time without modifying it.

[QPACK] provides additional details.

A.2.3. Flow Control Differences

HTTP/2 specifies a stream flow control mechanism. Although all HTTP/2 frames are delivered on streams, only the DATA frame payload is subject to flow control. QUIC provides flow control for stream data and all HTTP/3 frame types defined in this document are sent on streams. Therefore, all frame headers and payload are subject to flow control.

A.2.4. Guidance for New Frame Type Definitions

Frame type definitions in HTTP/3 often use the QUIC variable-length integer encoding. In particular, Stream IDs use this encoding, which allows for a larger range of possible values than the encoding used in HTTP/2. Some frames in HTTP/3 use an identifier other than a Stream ID (e.g., Push IDs). Redefinition of the encoding of extension frame types might be necessary if the encoding includes a Stream ID.

Because the Flags field is not present in generic HTTP/3 frames, those frames that depend on the presence of flags need to allocate space for flags as part of their frame payload.

Other than these issues, frame type HTTP/2 extensions are typically portable to QUIC simply by replacing Stream 0 in HTTP/2 with a control stream in HTTP/3. HTTP/3 extensions will not assume ordering, but would not be harmed by ordering, and are expected to be portable to HTTP/2.

A.2.5. Comparison Between HTTP/2 and HTTP/3 Frame Types

DATA (0x0): Padding is not defined in HTTP/3 frames. See Section 7.2.1.

HEADERS (0x1): The PRIORITY region of HEADERS is not defined in HTTP/3 frames. Padding is not defined in HTTP/3 frames. See Section 7.2.2.

PRIORITY (0x2): As described in Appendix A.2.1, HTTP/3 does not provide a means of signaling priority.

RST_STREAM (0x3): RST_STREAM frames do not exist in HTTP/3, since QUIC provides stream lifecycle management. The same code point is used for the CANCEL_PUSH frame (Section 7.2.3).

SETTINGS (0x4): SETTINGS frames are sent only at the beginning of the connection. See Section 7.2.4 and Appendix A.3.

PUSH_PROMISE (0x5): The PUSH_PROMISE frame does not reference a stream; instead the push stream references the PUSH_PROMISE frame using a Push ID. See Section 7.2.5.

PING (0x6): PING frames do not exist in HTTP/3, as QUIC provides equivalent functionality.

GOAWAY (0x7): GOAWAY does not contain an error code. In the client to server direction, it carries a Push ID instead of a server initiated stream ID. See Section 7.2.6.

WINDOW_UPDATE (0x8): WINDOW_UPDATE frames do not exist in HTTP/3, since QUIC provides flow control.

CONTINUATION (0x9): CONTINUATION frames do not exist in HTTP/3; instead, larger HEADERS/PUSH_PROMISE frames than HTTP/2 are permitted.

Frame types defined by extensions to HTTP/2 need to be separately registered for HTTP/3 if still applicable. The IDs of frames defined in [HTTP2] have been reserved for simplicity. Note that the frame type space in HTTP/3 is substantially larger (62 bits versus 8 bits), so many HTTP/3 frame types have no equivalent HTTP/2 code points. See Section 11.2.1.

A.3. HTTP/2 SETTINGS Parameters

An important difference from HTTP/2 is that settings are sent once, as the first frame of the control stream, and thereafter cannot change. This eliminates many corner cases around synchronization of changes.

Some transport-level options that HTTP/2 specifies via the SETTINGS frame are superseded by QUIC transport parameters in HTTP/3. The HTTP-level setting that is retained in HTTP/3 has the same value as in HTTP/2. The superseded settings are reserved, and their receipt is an error. See Section 7.2.4.1 for discussion of both the retained and reserved values.

Below is a listing of how each HTTP/2 SETTINGS parameter is mapped:

SETTINGS_HEADER_TABLE_SIZE (0x1): See [QPACK].

SETTINGS_ENABLE_PUSH (0x2): This is removed in favor of the MAX_PUSH_ID frame, which provides a more granular control over server push. Specifying a setting with the identifier 0x2 (corresponding to the SETTINGS_ENABLE_PUSH parameter) in the HTTP/3 SETTINGS frame is an error.

SETTINGS_MAX_CONCURRENT_STREAMS (0x3): QUIC controls the largest open Stream ID as part of its flow control logic. Specifying a setting with the identifier 0x3 (corresponding to the SETTINGS_MAX_CONCURRENT_STREAMS parameter) in the HTTP/3 SETTINGS frame is an error.

SETTINGS_INITIAL_WINDOW_SIZE (0x4): QUIC requires both stream and connection flow control window sizes to be specified in the initial transport handshake. Specifying a setting with the identifier 0x4 (corresponding to the SETTINGS_INITIAL_WINDOW_SIZE parameter) in the HTTP/3 SETTINGS frame is an error.

SETTINGS_MAX_FRAME_SIZE (0x5): This setting has no equivalent in HTTP/3. Specifying a setting with the identifier 0x5 (corresponding to the SETTINGS_MAX_FRAME_SIZE parameter) in the HTTP/3 SETTINGS frame is an error.

SETTINGS_MAX_HEADER_LIST_SIZE (0x6): This setting identifier has been renamed SETTINGS_MAX_FIELD_SECTION_SIZE.

In HTTP/3, setting values are variable-length integers (6, 14, 30, or 62 bits long) rather than fixed-length 32-bit fields as in HTTP/

2. This will often produce a shorter encoding, but can produce a longer encoding for settings that use the full 32-bit space. Settings ported from HTTP/2 might choose to redefine their value to limit it to 30 bits for more efficient encoding, or to make use of the 62-bit space if more than 30 bits are required.

Settings need to be defined separately for HTTP/2 and HTTP/3. The IDs of settings defined in [HTTP2] have been reserved for simplicity. Note that the settings identifier space in HTTP/3 is substantially larger (62 bits versus 16 bits), so many HTTP/3 settings have no equivalent HTTP/2 code point. See Section 11.2.2.

As QUIC streams might arrive out of order, endpoints are advised not to wait for the peers' settings to arrive before responding to other streams. See Section 7.2.4.2.

A.4. HTTP/2 Error Codes

QUIC has the same concepts of "stream" and "connection" errors that HTTP/2 provides. However, the differences between HTTP/2 and HTTP/3 mean that error codes are not directly portable between versions.

The HTTP/2 error codes defined in Section 7 of [HTTP2] logically map to the HTTP/3 error codes as follows:

NO_ERROR (0x0): H3_NO_ERROR in Section 8.1.

PROTOCOL_ERROR (0x1): This is mapped to H3_GENERAL_PROTOCOL_ERROR except in cases where more specific error codes have been defined. Such cases include H3_FRAME_UNEXPECTED, H3_MESSAGE_ERROR, and H3_CLOSED_CRITICAL_STREAM defined in Section 8.1.

INTERNAL_ERROR (0x2): H3_INTERNAL_ERROR in Section 8.1.

FLOW_CONTROL_ERROR (0x3): Not applicable, since QUIC handles flow control.

SETTINGS_TIMEOUT (0x4): Not applicable, since no acknowledgment of SETTINGS is defined.

STREAM_CLOSED (0x5): Not applicable, since QUIC handles stream management.

FRAME_SIZE_ERROR (0x6): H3_FRAME_ERROR error code defined in Section 8.1.

REFUSED_STREAM (0x7): H3_REQUEST_REJECTED (in Section 8.1) is used to indicate that a request was not processed. Otherwise, not applicable because QUIC handles stream management.

CANCEL (0x8):
H3_REQUEST_CANCELLED in Section 8.1.

COMPRESSION_ERROR (0x9): Multiple error codes are defined in [QPACK].

CONNECT_ERROR (0xa): H3_CONNECT_ERROR in Section 8.1.

ENHANCE_YOUR_CALM (0xb): H3_EXCESSIVE_LOAD in Section 8.1.

INADEQUATE_SECURITY (0xc): Not applicable, since QUIC is assumed to provide sufficient security on all connections.

HTTP_1_1_REQUIRED (0xd): H3_VERSION_FALLBACK in Section 8.1.

Error codes need to be defined for HTTP/2 and HTTP/3 separately. See Section 11.2.3.

A.4.1. Mapping Between HTTP/2 and HTTP/3 Errors

An intermediary that converts between HTTP/2 and HTTP/3 may encounter error conditions from either upstream. It is useful to communicate the occurrence of error to the downstream but error codes largely reflect connection-local problems that generally do not make sense to propagate.

An intermediary that encounters an error from an upstream origin can indicate this by sending an HTTP status code such as 502, which is suitable for a broad class of errors.

There are some rare cases where it is beneficial to propagate the error by mapping it to the closest matching error type to the receiver. For example, an intermediary that receives an HTTP/2 stream error of type REFUSED_STREAM from the origin has a clear signal that the request was not processed and that the request is safe to retry. Propagating this error condition to the client as an HTTP/3 stream error of type H3_REQUEST_REJECTED allows the client to take the action it deems most appropriate. In the reverse direction, the intermediary might deem it beneficial to pass on client request cancellations that are indicated by terminating a stream with H3_REQUEST_CANCELLED; see Section 4.1.2.

Conversion between errors is described in the logical mapping. The error codes are defined in non-overlapping spaces in order to protect against accidental conversion that could result in the use of inappropriate or unknown error codes for the target version. An intermediary is permitted to promote stream errors to connection errors but they should be aware of the cost to the HTTP/3 connection for what might be a temporary or intermittent error.

Appendix B. Change Log

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

B.1. Since draft-ietf-quic-http-32

* Removed draft version guidance; added final version string

* Added H3_MESSAGE_ERROR for malformed messages

B.2. Since draft-ietf-quic-http-31

Editorial changes only.

B.3. Since draft-ietf-quic-http-30

Editorial changes only.

B.4. Since draft-ietf-quic-http-29

* Require a connection error if a reserved frame type that corresponds to a frame in HTTP/2 is received (#3991, #3993)

* Require a connection error if a reserved setting that corresponds to a setting in HTTP/2 is received (#3954, #3955)

B.5. Since draft-ietf-quic-http-28

* CANCEL_PUSH is recommended even when the stream is reset (#3698, #3700)

* Use H3_ID_ERROR when GOAWAY contains a larger identifier (#3631, #3634)

B.6. Since draft-ietf-quic-http-27

* Updated text to refer to latest HTTP revisions

* Use the HTTP definition of authority for establishing and coalescing connections (#253, #2223, #3558)

* Define use of GOAWAY from both endpoints (#2632, #3129)

* Require either :authority or Host if the URI scheme has a mandatory authority component (#3408, #3475)

B.7. Since draft-ietf-quic-http-26

* No changes

B.8. Since draft-ietf-quic-http-25

* Require QUICv1 for HTTP/3 (#3117, #3323)

* Remove DUPLICATE_PUSH and allow duplicate PUSH_PROMISE (#3275, #3309)

* Clarify the definition of "malformed" (#3352, #3345)

B.9. Since draft-ietf-quic-http-24

* Removed H3_EARLY_RESPONSE error code; H3_NO_ERROR is recommended instead (#3130,#3208)

* Unknown error codes are equivalent to H3_NO_ERROR (#3276,#3331)

* Some error codes are reserved for greasing (#3325,#3360)

B.10. Since draft-ietf-quic-http-23

* Removed quic Alt-Svc parameter (#3061,#3118)

* Clients need not persist unknown settings for use in 0-RTT (#3110,#3113)

* Clarify error cases around CANCEL_PUSH (#2819,#3083)

B.11. Since draft-ietf-quic-http-22

* Removed priority signaling (#2922,#2924)

* Further changes to error codes (#2662,#2551):

  - Error codes renumbered

  - HTTP_MALFORMED_FRAME replaced by HTTP_FRAME_ERROR, HTTP_ID_ERROR, and others

* Clarify how unknown frame types interact with required frame sequence (#2867,#2858)

* Describe interactions with the transport in terms of defined interface terms (#2857,#2805)

* Require the use of the http-opportunistic resource (RFC 8164) when scheme is http (#2439,#2973)

* Settings identifiers cannot be duplicated (#2979)

* Changes to SETTINGS frames in 0-RTT (#2972,#2790,#2945):

  - Servers must send all settings with non-default values in their SETTINGS frame, even when resuming

  - If a client doesn't have settings associated with a 0-RTT ticket, it uses the defaults

  - Servers can't accept early data if they cannot recover the settings the client will have remembered

* Clarify that Upgrade and the 101 status code are prohibited (#2898,#2889)

* Clarify that frame types reserved for greasing can occur on any stream, but frame types reserved due to HTTP/2 correspondence are prohibited (#2997,#2692,#2693)

* Unknown error codes cannot be treated as errors (#2998,#2816)

B.12. Since draft-ietf-quic-http-21

No changes

B.13. Since draft-ietf-quic-http-20

* Prohibit closing the control stream (#2509, #2666)

* Change default priority to use an orphan node (#2502, #2690)

* Exclusive priorities are restored (#2754, #2781)

* Restrict use of frames when using CONNECT (#2229, #2702)

* Close and maybe reset streams if a connection error occurs for CONNECT (#2228, #2703)

* Encourage provision of sufficient unidirectional streams for QPACK (#2100, #2529, #2762)

* Allow extensions to use server-initiated bidirectional streams (#2711, #2773)

* Clarify use of maximum header list size setting (#2516, #2774)

* Extensive changes to error codes and conditions of their sending

  - Require connection errors for more error conditions (#2511, #2510)

  - Updated the error codes for illegal GOAWAY frames (#2714, #2707)

  - Specified error code for HEADERS on control stream (#2708)

  - Specified error code for servers receiving PUSH_PROMISE (#2709)

  - Specified error code for receiving DATA before HEADERS (#2715)

  - Describe malformed messages and their handling (#2410, #2764)

  - Remove HTTP_PUSH_ALREADY_IN_CACHE error (#2812, #2813)

  - Refactor Push ID related errors (#2818, #2820)

  - Rationalize HTTP/3 stream creation errors (#2821, #2822)

B.14. Since draft-ietf-quic-http-19

* SETTINGS_NUM_PLACEHOLDERS is 0x9 (#2443,#2530)

* Non-zero bits in the Empty field of the PRIORITY frame MAY be treated as an error (#2501)

B.15. Since draft-ietf-quic-http-18

* Resetting streams following a GOAWAY is recommended, but not required (#2256,#2457)

* Use variable-length integers throughout (#2437,#2233,#2253,#2275)

  - Variable-length frame types, stream types, and settings identifiers

  - Renumbered stream type assignments

  - Modified associated reserved values

* Frame layout switched from Length-Type-Value to Type-Length-Value (#2395,#2235)

* Specified error code for servers receiving DUPLICATE_PUSH (#2497)

* Use connection error for invalid PRIORITY (#2507, #2508)

B.16. Since draft-ietf-quic-http-17

* HTTP_REQUEST_REJECTED is used to indicate a request can be retried (#2106, #2325)

* Changed error code for GOAWAY on the wrong stream (#2231, #2343)

B.17. Since draft-ietf-quic-http-16

* Rename "HTTP/QUIC" to "HTTP/3" (#1973)

* Changes to PRIORITY frame (#1865, #2075)

  - Permitted as first frame of request streams

  - Remove exclusive reprioritization

  - Changes to Prioritized Element Type bits

* Define DUPLICATE_PUSH frame to refer to another PUSH_PROMISE (#2072)

* Set defaults for settings, allow request before receiving SETTINGS (#1809, #1846, #2038)

* Clarify message processing rules for streams that aren't closed (#1972, #2003)

* Removed reservation of error code 0 and moved HTTP_NO_ERROR to this value (#1922)

* Removed prohibition of zero-length DATA frames (#2098)

B.18. Since draft-ietf-quic-http-15

Substantial editorial reorganization; no technical changes.

B.19. Since draft-ietf-quic-http-14

* Recommend sensible values for QUIC transport parameters (#1720,#1806)

* Define error for missing SETTINGS frame (#1697,#1808)

* Setting values are variable-length integers (#1556,#1807) and do not have separate maximum values (#1820)

* Expanded discussion of connection closure (#1599,#1717,#1712)

* HTTP_VERSION_FALLBACK falls back to HTTP/1.1 (#1677,#1685)

B.20. Since draft-ietf-quic-http-13

* Reserved some frame types for grease (#1333, #1446)

* Unknown unidirectional stream types are tolerated, not errors; some reserved for grease (#1490, #1525)

* Require settings to be remembered for 0-RTT, prohibit reductions (#1541, #1641)

* Specify behavior for truncated requests (#1596, #1643)

B.21. Since draft-ietf-quic-http-12

* TLS SNI extension isn't mandatory if an alternative method is used (#1459, #1462, #1466)

* Removed flags from HTTP/3 frames (#1388, #1398)

* Reserved frame types and settings for use in preserving extensibility (#1333, #1446)

* Added general error code (#1391, #1397)

* Unidirectional streams carry a type byte and are extensible (#910,#1359)

* Priority mechanism now uses explicit placeholders to enable persistent structure in the tree (#441,#1421,#1422)

B.22. Since draft-ietf-quic-http-11

* Moved QPACK table updates and acknowledgments to dedicated streams (#1121, #1122, #1238)

B.23. Since draft-ietf-quic-http-10

* Settings need to be remembered when attempting and accepting 0-RTT (#1157, #1207)

B.24. Since draft-ietf-quic-http-09

* Selected QCRAM for header compression (#228, #1117)

* The server_name TLS extension is now mandatory (#296, #495)

* Specified handling of unsupported versions in Alt-Svc (#1093, #1097)

B.25. Since draft-ietf-quic-http-08

* Clarified connection coalescing rules (#940, #1024)

B.26. Since draft-ietf-quic-http-07

* Changes for integer encodings in QUIC (#595,#905)
* Use unidirectional streams as appropriate (#515, #240, #281, #886)
* Improvement to the description of GOAWAY (#604, #898)
* Improve description of server push usage (#947, #950, #957)

B.27. Since draft-ietf-quic-http-06

* Track changes in QUIC error code usage (#485)

B.28. Since draft-ietf-quic-http-05

* Made push ID sequential, add MAX_PUSH_ID, remove SETTINGS_ENABLE_PUSH (#709)
* Guidance about keep-alive and QUIC PINGs (#729)
* Expanded text on GOAWAY and cancellation (#757)

B.29. Since draft-ietf-quic-http-04

* Cite RFC 5234 (#404)
* Return to a single stream per request (#245,#557)
* Use separate frame type and settings registries from HTTP/2 (#81)
* SETTINGS_ENABLE_PUSH instead of SETTINGS_DISABLE_PUSH (#477)
* Restored GOAWAY (#696)
* Identify server push using Push ID rather than a stream ID (#702,#281)
* DATA frames cannot be empty (#700)

B.30. Since draft-ietf-quic-http-03

None.

B.31. Since draft-ietf-quic-http-02

* Track changes in transport draft

B.32. Since draft-ietf-quic-http-01

* SETTINGS changes (#181):

  - SETTINGS can be sent only once at the start of a connection; no changes thereafter

  - SETTINGS_ACK removed

  - Settings can only occur in the SETTINGS frame a single time

  - Boolean format updated

* Alt-Svc parameter changed from "v" to "quic"; format updated (#229)

* Closing the connection control stream or any message control stream is a fatal error (#176)

* HPACK Sequence counter can wrap (#173)

* 0-RTT guidance added

* Guide to differences from HTTP/2 and porting HTTP/2 extensions added (#127,#242)

B.33. Since draft-ietf-quic-http-00

* Changed "HTTP/2-over-QUIC" to "HTTP/QUIC" throughout (#11,#29)

* Changed from using HTTP/2 framing within Stream 3 to new framing format and two-stream-per-request model (#71,#72,#73)

* Adopted SETTINGS format from draft-bishop-httpbis-extended-settings-01

* Reworked SETTINGS_ACK to account for indeterminate inter-stream order (#75)

* Described CONNECT pseudo-method (#95)

* Updated ALPN token and Alt-Svc guidance (#13,#87)

* Application-layer-defined error codes (#19,#74)

B.34. Since draft-shade-quic-http2-mapping-00

* Adopted as base for draft-ietf-quic-http

* Updated authors/editors list

Acknowledgments

The original authors of this specification were Robbie Shade and Mike Warres.

The IETF QUIC Working Group received an enormous amount of support from many people. Among others, the following people provided substantial contributions to this document:

* Bence Beky
* Daan De Meyer
* Martin Duke
* Roy Fielding
* Alan Frindell
* Alessandro Ghedini
* Nick Harper
* Ryan Hamilton
* Christian Huitema
* Subodh Iyengar
* Robin Marx
* Patrick McManus
* Luca Niccolini
* (Kazuho Oku)
* Lucas Pardue
* Roberto Peon
* Julian Reschke
* Eric Rescorla
* Martin Seemann
* Ben Schwartz
* Ian Swett

*Willy Taureau

*Martin Thomson

*Dmitri Tikhonov

*Tatsuhiro Tsujikawa

A portion of Mike's contribution was supported by Microsoft during his employment there.

Author's Address

Mike Bishop (editor)
Akamai

Email: mbishop@evequefou.be