Hypertext Transfer Protocol Version 3 (HTTP/3)
draft-ietf-quic-http-22

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.

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 [1].

   Working Group information can be found at https://github.com/quicwg
   [2]; source code and issues list for this draft can be found at
   https://github.com/quicwg/base-drafts/labels/-http [3].

Table of Contents

1. Introduction

   HTTP semantics are used for a broad range of services on the
   Internet. These semantics have commonly been used with two different
   TCP mappings, HTTP/1.1 and HTTP/2. HTTP/3 supports the same
   semantics over a new transport protocol, QUIC.

1.1. Prior versions of HTTP

   HTTP/1.1 is a TCP mapping which uses whitespace-delimited text fields
   to convey HTTP messages. While these exchanges are human-readable,
   using whitespace for message formatting leads to parsing difficulties
   and workarounds to be tolerant of variant behavior. Because each
   connection can transfer only a single HTTP request or response at a
   time in each direction, multiple parallel TCP connections are often
   used, reducing the ability of the congestion controller to accurately
   manage traffic between endpoints.

   HTTP/2 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 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, it has the capability to
improve the performance of HTTP compared to a TCP mapping. QUIC also
incorporates TLS 1.3 at the transport layer, offering comparable
security to running TLS over TCP, with the improved connection setup
latency of TCP Fast Open [RFC7413]}.

This document defines a mapping of HTTP semantics over the QUIC
transport protocol, drawing heavily on the design of HTTP/2. While
delegating stream lifetime and flow control issues to QUIC, a similar
binary 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. An HTTP/3
endpoint can be discovered using HTTP Alternative Services; this
process is described in greater detail in Section 3.2.

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). Other frame types like SETTINGS, PRIORITY,
and GOAWAY are used to manage the overall connection and
relationships between streams.

Multiplexing of requests is performed using the QUIC stream
abstraction, described in Section 2 of [QUIC-TRANSPORT]. Each
request and response 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] which
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,
DUPLICATE_PUSH, MAX_PUSH_ID, and CANCEL_PUSH.

As in HTTP/2, request and response headers are compressed for
transmission. Because HPACK [HPACK] relies on in-order transmission

of compressed header blocks (a guarantee not provided by QUIC),
HTTP/3 replaces HPACK with QPACK [QPACK]. QPACK uses separate
unidirectional streams to modify and track header table state, while
header blocks refer to the state of the table without modifying it.

2.1. Document Organization

The HTTP/3 specification is split into seven parts. The document
begins with a detailed overview of the connection lifecycle and key
concepts:

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

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

o Connection Closure (Section 5) describes how connections are
   terminated, either gracefully or abruptly.

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

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

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

o 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:

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

o 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.

Field definitions are given in Augmented Backus-Naur Form (ABNF), as
defined in [RFC5234].

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 octets
   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.

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.

stream error: An error on the individual HTTP/3 stream.

The term "payload body" is defined in Section 3.3 of [RFC7230].

Finally, the terms "gateway", "intermediary", "proxy", and "tunnel"
are defined in Section 2.3 of [RFC7230]. Intermediaries act as both
client and server at different times.

3. Connection Setup and Management

3.1. Draft Version Identification

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

HTTP/3 uses the token "h3" to identify itself in ALPN and Alt-Svc.
Only implementations of the final, published RFC can identify
themselves as "h3". Until such an RFC exists, implementations MUST
NOT identify themselves using this string.

Implementations of draft versions of the protocol MUST add the string
"-" and the corresponding draft number to the identifier. For
example, draft-ietf-quic-http-01 is identified using the string
"h3-01".

Non-compatible experiments that are based on these draft versions
MUST append the string "-" and an experiment name to the identifier.
For example, an experimental implementation based on draft-ietf-quic-
http-09 which reserves an extra stream for unsolicited transmission
of 1980s pop music might identify itself as "h3-09-rickroll". Note
that any label MUST conform to the "token" syntax defined in
Section 3.2.6 of [RFC7230]. Experimenters are encouraged to
coordinate their experiments on the quic@ietf.org mailing list.

3.2. Discovering an HTTP/3 Endpoint

An HTTP origin advertises 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 ALPN token defined in Section 3.3.

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 and, if successful, send HTTP requests using the mapping
described in this document.

Connectivity problems (e.g. firewall blocking UDP) can result in QUIC connection establishment failure, in which case the client SHOULD continue using the existing connection or try another alternative endpoint offered by the origin.

Servers MAY serve HTTP/3 on any UDP port, since an alternative always includes an explicit port.

3.2.1. QUIC Version Hints

This document defines the "quic" parameter for Alt-Svc, which MAY be used to provide version-negotiation hints to HTTP/3 clients. QUIC versions are four-byte sequences with no additional constraints on format. Leading zeros SHOULD be omitted for brevity.

Syntax:

quic = DQUOTE version-number ["," version-number] * DQUOTE
version-number = 1*8HEXDIG; hex-encoded QUIC version

Where multiple versions are listed, the order of the values reflects the server's preference (with the first value being the most preferred version). Reserved versions MAY be listed, but unreserved versions which are not supported by the alternative SHOULD NOT be present in the list. Origins MAY omit supported versions for any reason.

Clients MUST ignore any included versions which they do not support. The "quic" parameter MUST NOT occur more than once; clients SHOULD process only the first occurrence.

For example, suppose a server supported both version 0x00000001 and the version rendered in ASCII as "Q034". If it also opted to include the reserved version (from Section 15 of [QUIC-TRANSPORT]) 0x1abadaba, it could specify the following header field:

Alt-Svc: h3=":49288";quic="1,1abadaba,51303334"

A client acting on this header field would drop the reserved version (not supported), then attempt to connect to the alternative using the first version in the list which it does support, if any.

3.3. Connection Establishment

HTTP/3 relies on QUIC as the underlying transport. The QUIC version being used MUST use TLS version 1.3 or greater as its handshake protocol. HTTP/3 clients MUST indicate the target domain name during

the TLS handshake. This may be done using the Server Name Indication
(SNI) [RFC6066] extension to TLS or using some other mechanism.

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 7.2.5) MUST be sent by each
endpoint as the initial frame of their respective HTTP control stream
(see Section 6.2.1).

3.4. Connection Reuse

Once a connection exists to a server endpoint, this connection MAY be
reused for requests with multiple different URI authority components.
The client MAY send any requests for which the client considers the
server authoritative.

An authoritative HTTP/3 endpoint is typically discovered because the
client has received an Alt-Svc record from the request's origin which
nominates the endpoint as a valid HTTP Alternative Service for that
origin. As required by [RFC7838], clients MUST check that the
nominated server can present a valid certificate for the origin
before considering it authoritative. Clients MUST NOT assume that an
HTTP/3 endpoint is authoritative for other origins without an
explicit signal.

A server that does not wish clients to reuse 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 9.1.2 of [HTTP2]).

The considerations discussed in Section 9.1 of [HTTP2] also apply to
the management of HTTP/3 connections.

4. HTTP Request Lifecycle

4.1. HTTP Message Exchanges

A client sends an HTTP request on a client-initiated bidirectional
QUIC stream. A client MUST send only a single request on a given
stream. A server sends zero or more non-final HTTP responses on the

same stream as the request, followed by a single final HTTP response,
as detailed below.

An HTTP message (request or response) consists of:

1. the message header (see [RFC7230], Section 3.2), sent as a single
    HEADERS frame (see Section 7.2.2),

2. the payload body (see [RFC7230], Section 3.3), sent as a series
    of DATA frames (see Section 7.2.1),

3. optionally, one HEADERS frame containing the trailer-part, if
    present (see [RFC7230], Section 4.1.2).

A server MAY send one or more PUSH_PROMISE frames (see Section 7.2.6)
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.

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.

The "chunked" transfer encoding defined in Section 4.1 of [RFC7230]
MUST NOT be used.

If a DATA frame is received before a HEADERS frame on a either a
request or push stream, the recipient MUST respond with a connection
error of type HTTP_UNEXPECTED_FRAME (Section 8).

Trailing header fields are carried in an additional HEADERS frame
following the body. Senders MUST send only one HEADERS frame in the
trailers section; receivers MUST discard any subsequent HEADERS
frames.

A response MAY consist of multiple messages when and only when one or
more informational responses (1xx; see [RFC7231], Section 6.2)
precede a final response to the same request. Non-final responses do
not contain a payload body or trailers.

An HTTP request/response exchange fully consumes a 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 an 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 stream terminates without
enough of the HTTP message to provide a complete response, the server
SHOULD abort its response with the error code
HTTP_INCOMPLETE_REQUEST.

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 this is true, a
server MAY request that the client abort transmission of a request
without error by triggering a QUIC STOP_SENDING frame with error code
HTTP_EARLY_RESPONSE, sending a complete response, and cleanly closing
its 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.

4.1.1. Header Formatting and Compression

HTTP message headers carry information as a series of key-value
pairs, called header fields. For a listing of registered HTTP header
fields, see the "Message Header Field" registry maintained at
https://www.iana.org/assignments/message-headers [4].

Just as in previous versions of HTTP, header field names are strings
of ASCII characters that are compared in a case-insensitive fashion.
Properties of HTTP header field names and values are discussed in
more detail in Section 3.2 of [RFC7230], though the wire rendering in
HTTP/3 differs. As in HTTP/2, header field names MUST be converted
to lowercase prior to their encoding. A request or response
containing uppercase header field names MUST be treated as malformed
(Section 4.1.3).

As in HTTP/2, HTTP/3 uses special pseudo-header fields beginning with
the ':' character (ASCII 0x3a) to convey the target URI, the method
of the request, and the status code for the response. These pseudo-
header fields are defined in Section 8.1.2.3 and 8.1.2.4 of [HTTP2].
Pseudo-header fields are not HTTP header fields. Endpoints MUST NOT
generate pseudo-header fields other than those defined in [HTTP2].
The restrictions on the use of pseudo-header fields in
Section 8.1.2.1 of [HTTP2] also apply to HTTP/3.

HTTP/3 uses QPACK header compression as described in [QPACK], a
variation of HPACK which allows the flexibility to avoid header-
compression-induced head-of-line blocking. See that document for
additional details.

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 field list 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 header field list is calculated based on the
uncompressed size of header fields, including the length of the name
and value in bytes plus an overhead of 32 bytes for each header
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_HEADER_LIST_SIZE" parameter. An implementation which
has received this parameter SHOULD NOT send an HTTP message header
which exceeds the indicated size, as the peer will likely refuse to
process it. However, because this limit is applied at each hop,
messages below this limit are not guaranteed to be accepted.

4.1.2. Request Cancellation and Rejection

Clients can cancel requests by aborting the stream (QUIC RESET_STREAM
and/or STOP_SENDING frames, as appropriate) with an error code of
HTTP_REQUEST_CANCELLED (Section 8.1). When the client cancels a
response, it indicates that this response is no longer of interest.
Implementations SHOULD cancel requests by aborting both directions of
a stream.

When the server rejects a request without performing any application
processing, it SHOULD abort its response stream with the error code
HTTP_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 on a new connection.
Servers MUST NOT use the HTTP_REQUEST_REJECTED error code for
requests which were partially or fully processed. When a server
abandons a response after partial processing, it SHOULD abort its
response stream with the error code HTTP_REQUEST_CANCELLED.

When a client sends a STOP_SENDING with HTTP_REQUEST_CANCELLED, a
server MAY send the error code HTTP_REQUEST_REJECTED in the
corresponding RESET_STREAM if no processing was performed. Clients
MUST NOT reset streams with the HTTP_REQUEST_REJECTED error code
except in response to a QUIC STOP_SENDING frame that contains the
same code.

If a stream is cancelled 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. Automatically retrying such requests is
not possible, unless this is otherwise permitted (e.g., idempotent
actions like GET, PUT, or DELETE).

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 extraneous
frames, prohibited header fields, the absence of mandatory header
fields, or the inclusion of uppercase header field names.

A request or response that includes a payload body can include a
"content-length" header field. A request or response is also
malformed if the value of a content-length header field does not
equal the sum of the DATA frame payload lengths that form the body.
A response that is defined to have no payload, as described in
Section 3.3.2 of [RFC7230] can have a non-zero content-length header
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
HTTP_GENERAL_PROTOCOL_ERROR.

For malformed requests, a server MAY send an HTTP response 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 pseudo-method CONNECT ([RFC7231], Section 4.3.6) 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, the CONNECT method is used to
establish a tunnel over a single HTTP/2 stream to a remote host for
similar purposes.

A CONNECT request in HTTP/3 functions in the same manner as in
HTTP/2. The request MUST be formatted as described in [HTTP2],
Section 8.3. A CONNECT request that does not conform to these
restrictions is malformed (see Section 4.1.3). The request stream
MUST NOT be closed at the end of the request.

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 [RFC7231], Section 4.3.6.

All DATA frames on the stream correspond to data sent or received on
the TCP connection. 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 frame type MUST be treated as a connection error of type
HTTP_UNEXPECTED_FRAME.

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 terminate the send stream that it sends to
the client. TCP connections which 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 with QUIC RESET_STREAM frame. 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 HTTP_CONNECT_ERROR (Section 8.1). 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.

4.3. Prioritization

The purpose of prioritization is to allow a client to express how it
would prefer the server to allocate resources when managing
concurrent streams. Most importantly, priority can be used to select
streams for transmitting frames when there is limited capacity for
sending.

HTTP/3 uses a priority scheme similar to that described in [RFC7540],
Section 5.3. In this priority scheme, a given element can be
designated as dependent upon another element. Each dependency is

   assigned a relative weight, a number that is used to determine the
   relative proportion of available resources that are assigned to
   streams dependent on the same stream. This information is expressed
   in the PRIORITY frame Section 7.2.3 which identifies the element and
   the dependency. The elements that can be prioritized are:

   o Requests, identified by the ID of the request stream

   o Pushes, identified by the Push ID of the promised resource
      (Section 7.2.6)

   o Placeholders, identified by a Placeholder ID

   Taken together, the dependencies across all prioritized elements in a
   connection form a dependency tree. An element can depend on another
   element or on the root of the tree. The tree also contains an orphan
   placeholder. This placeholder cannot be reprioritized, and no
   resources should be allocated to descendants of the orphan
   placeholder if progress can be made on descendants of the root. The
   structure of the dependency tree changes as PRIORITY frames modify
   the dependency links between other prioritized elements.

   An exclusive flag allows for the insertion of a new level of
   dependencies. The exclusive flag causes the prioritized element to
   become the sole dependency of its parent, causing other dependencies
   to become dependent on the exclusive element.

   All dependent streams are allocated an integer weight between 1 and
   256 (inclusive), derived by adding one to the weight expressed in the
   PRIORITY frame.

   Streams with the same parent SHOULD be allocated resources
   proportionally based on their weight. Thus, if stream B depends on
   stream A with weight 4, stream C depends on stream A with weight 12,
   and no progress can be made on stream A, stream B ideally receives
   one-third of the resources allocated to stream C.

   A reference to an element which is no longer in the tree is treated
   as a reference to the orphan placeholder. Due to reordering between
   streams, an element can also be prioritized which is not yet in the
   tree. Such elements are added to the tree with the requested
   priority. If a prioritized element depends on another element which
   is not yet in the tree, the requested parent is first added to the
   tree with the default priority.

   When a prioritized element is first created, it has a default initial
   weight of 16 and a default dependency. Requests and placeholders are

dependent on the orphan placeholder; pushes are dependent on the
client request on which the PUSH_PROMISE frame was sent.

Priorities can be updated by sending a PRIORITY frame (see
Section 7.2.3) on the control stream.

4.3.1. Placeholders

In HTTP/2, certain implementations used closed or unused streams as
placeholders in describing the relative priority of requests. This
created confusion as servers could not reliably identify which
elements of the priority tree could be discarded safely. Clients
could potentially reference closed streams long after the server had
discarded state, leading to disparate views of the prioritization the
client had attempted to express.

In HTTP/3, a number of placeholders are explicitly permitted by the
server using the "SETTINGS_NUM_PLACEHOLDERS" setting. Because the
server commits to maintaining these placeholders in the
prioritization tree, clients can use them with confidence that the
server will not have discarded the state. Clients MUST NOT send the
"SETTINGS_NUM_PLACEHOLDERS" setting; receipt of this setting by a
server MUST be treated as a connection error of type
"HTTP_SETTINGS_ERROR".

Client-controlled placeholders are identified by an ID between zero
and one less than the number of placeholders the server has
permitted. The orphan placeholder cannot be prioritized or
referenced by the client.

Like streams, client-controlled placeholders have priority
information associated with them.

4.3.2. Priority Tree Maintenance

Because placeholders will be used to "root" any persistent structure
of the tree which the client cares about retaining, servers can
aggressively prune inactive regions from the priority tree. For
prioritization purposes, a node in the tree is considered "inactive"
when the corresponding stream has been closed for at least two round-
trip times (using any reasonable estimate available on the server).
This delay helps mitigate race conditions where the server has pruned
a node the client believed was still active and used as a Stream
Dependency.

Specifically, the server MAY at any time:

o Identify and discard branches of the tree containing only inactive
   nodes (i.e. a node with only other inactive nodes as descendants,
   along with those descendants)

o Identify and condense interior regions of the tree containing only
   inactive nodes, allocating weight appropriately

```
   X               X               X
   |               |               |
   P               P               P
  / \              |               |
 I   I    ==>      I      ==>       A
    / \            |                |
   A   I           A                A
   |               |
   A               A
```
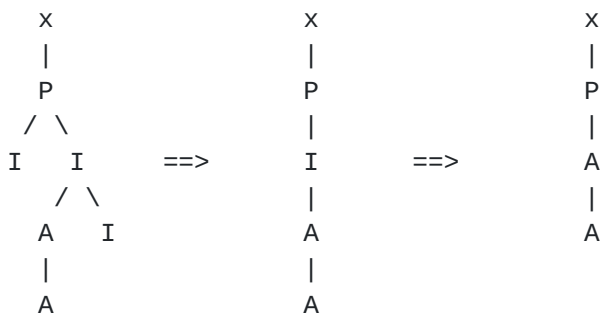
            Figure 1: Example of Priority Tree Pruning

In the example in Figure 1, "P" represents a Placeholder, "A"
represents an active node, and "I" represents an inactive node. In
the first step, the server discards two inactive branches (each a
single node). In the second step, the server condenses an interior
inactive node. Note that these transformations will result in no
change in the resources allocated to a particular active stream.

Clients SHOULD assume the server is actively performing such pruning
and SHOULD NOT declare a dependency on a stream it knows to have been
closed.

4.4. Server Push

Server push is an interaction mode introduced in HTTP/2 [HTTP2] which
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 HTTP/2 [HTTP2], but uses
different mechanisms.

Each server push is identified by a unique Push ID. This Push ID is
used in a single PUSH_PROMISE frame (see Section 7.2.6) which carries
the request headers, possibly included in one or more DUPLICATE_PUSH
frames (see Section 7.2.9), then included with the push stream which
ultimately fulfills those promises.

Server push is only enabled on a connection when a client sends a
MAX_PUSH_ID frame (see Section 7.2.8). A server cannot use server
push until it receives a MAX_PUSH_ID frame. A client sends

additional MAX_PUSH_ID frames to control the number of pushes that a
server can promise. A server SHOULD use Push IDs sequentially,
starting at 0. A client MUST treat receipt of a push stream with a
Push ID that is greater than the maximum Push ID as a connection
error of type HTTP_ID_ERROR.

The header of the request message is carried by a PUSH_PROMISE frame
(see Section 7.2.6) on the request stream which generated the push.
This allows the server push to be associated with a client request.
Promised requests MUST conform to the requirements in Section 8.2 of
[HTTP2].

The same server push can be associated with additional client
requests using a DUPLICATE_PUSH frame (see Section 7.2.9).

Ordering of a PUSH_PROMISE or DUPLICATE_PUSH in relation to certain
parts of the response is important. The server SHOULD send
PUSH_PROMISE or DUPLICATE_PUSH 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.

When a server later fulfills a promise, the server push response is
conveyed on a push stream (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 using the same format described
for responses in Section 4.1.

Due to reordering, DUPLICATE_PUSH frames or push stream data can
arrive before the corresponding PUSH_PROMISE frame. When a client
receives a DUPLICATE_PUSH frame for an as-yet-unknown Push ID, the
request headers of the push are not immediately available. The
client can either delay generating new requests for content
referenced following the DUPLICATE_PUSH frame until the request
headers become available, or can initiate requests for discovered
resources and cancel the requests if the requested resource is
already being pushed. When a client receives a new push stream with
an as-yet-unknown Push ID, both the associated client request and the
pushed request headers 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.

If a promised server push is not needed by the client, the client
SHOULD send a CANCEL_PUSH frame. If the push stream is already open
or opens after sending the CANCEL_PUSH frame, a QUIC STOP_SENDING
frame with an error code of HTTP_REQUEST_CANCELLED can be used. This

asks the server not to transfer additional data and indicates that it
will be discarded upon receipt.

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 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 connection
for new requests if the existing connection has been idle for longer
than the server's advertised idle timeout, and SHOULD do so if
approaching the idle timeout.

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 19.2 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 let the connection close gracefully.
Since clients drive request generation, clients perform a connection
shutdown by not sending additional requests on the connection;
responses and pushed responses associated to previous requests will
continue to completion. Servers perform the same function by
communicating with clients.

Servers initiate the shutdown of a connection by sending a GOAWAY
frame (Section 7.2.7). The GOAWAY frame indicates that client-
initiated requests on lower stream IDs were or might be processed in
this connection, while requests on the indicated stream ID and
greater were rejected. This enables client and server to agree on
which requests were accepted prior to the connection shutdown. This
identifier MAY be zero if no requests were processed. Servers SHOULD
NOT increase the QUIC MAX_STREAMS limit after sending a GOAWAY frame.

Clients MUST NOT send new requests on the connection after receiving
GOAWAY; a new connection MAY be established to send additional
requests.

Some requests might already be in transit. If the client has already
sent requests on streams with a Stream ID greater than or equal to
that indicated in the GOAWAY frame, those requests will not be
processed and MAY be retried by the client on a different connection.
The client MAY cancel these requests. It is RECOMMENDED that the
server explicitly reject such requests (see Section 4.1.2) in order
to clean up transport state for the affected streams.

Requests on Stream IDs less than the Stream ID in the GOAWAY frame
might have been processed; their status cannot be known until a
response is received, the stream is reset individually, or the
connection terminates. Servers MAY reject individual requests on
streams below the indicated ID if these requests were not processed.

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.

A client that is unable to retry requests loses all requests that are
in flight when the server closes the connection. A server MAY send
multiple GOAWAY frames indicating different stream IDs, but MUST NOT
increase the value they send in the last Stream ID, since clients
might already have retried unprocessed requests on another
connection. A server that is attempting to gracefully shut down a
connection SHOULD send an initial GOAWAY frame with the last Stream
ID set to the maximum value allowed by QUIC's MAX_STREAMS and SHOULD
NOT increase the MAX_STREAMS limit thereafter. This signals to the
client that a shutdown is imminent and that initiating further
requests is prohibited. After allowing time for any in-flight
requests (at least one round-trip time), the server MAY send another
GOAWAY frame with an updated last Stream ID. This ensures that a
connection can be cleanly shut down without losing requests.

Once all accepted requests have been processed, the server 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 HTTP_NO_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; the error code in this frame indicates to the peer why the
connection is being closed. See Section 8 for error codes which can
be used when closing a connection.

Before closing the connection, a GOAWAY 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.

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 which interrupts connectivity.

If a connection terminates without a GOAWAY frame, clients MUST
assume that any request which 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. On the wire, data is framed into QUIC STREAM frames, but
this framing is invisible to the HTTP framing layer. The transport
layer buffers and orders received QUIC STREAM frames, exposing the
data contained within as 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 headers 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 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. 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 client SHOULD send non-zero values for the QUIC transport
   parameters "initial_max_stream_data_bidi_local". An HTTP/3 server
   SHOULD send non-zero values for the QUIC transport parameters
   "initial_max_stream_data_bidi_remote" and "initial_max_bidi_streams".
   It is RECOMMENDED that "initial_max_bidi_streams" be no smaller than
   100, so as to not unnecessarily limit parallelism.

   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 HTTP_STREAM_CREATION_ERROR 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.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Stream Type (i)                      ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```
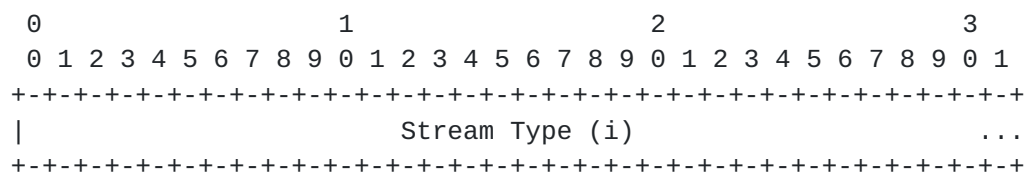
                   Figure 2: Unidirectional Stream Header

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

   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 set low values for the QUIC
   transport parameters "initial_max_uni_streams" and
   "initial_max_stream_data_uni" will increase the chance that the
   remote peer reaches the limit early and becomes blocked. In
   particular, the value chosen for "initial_max_uni_streams" should

   consider that remote peers may wish to exercise reserved stream
   behavior (Section 6.2.3). To avoid blocking, 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 (such as QPACK) by setting
   an appropriate value for the QUIC transport parameter
   "initial_max_uni_streams" (three being the minimum value required for
   the base HTTP/3 protocol and QPACK), and SHOULD use a value of 1,024
   or greater for the QUIC transport parameter
   "initial_max_stream_data_uni".

   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 which 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
   trigger a QUIC STOP_SENDING frame with an error code of
   HTTP_STREAM_CREATION_ERROR, 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 which 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
   HTTP_MISSING_SETTINGS. Only one control stream per peer is
   permitted; receipt of a second stream which claims to be a control
   stream MUST be treated as a connection error of type

HTTP_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
HTTP_CLOSED_CRITICAL_STREAM.

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 enabled on the
connection, either client or server might be able to send stream data
first after the cryptographic handshake completes.

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. 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
HTTP_STREAM_CREATION_ERROR.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          0x01 (i)                          ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Push ID (i)                        ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```
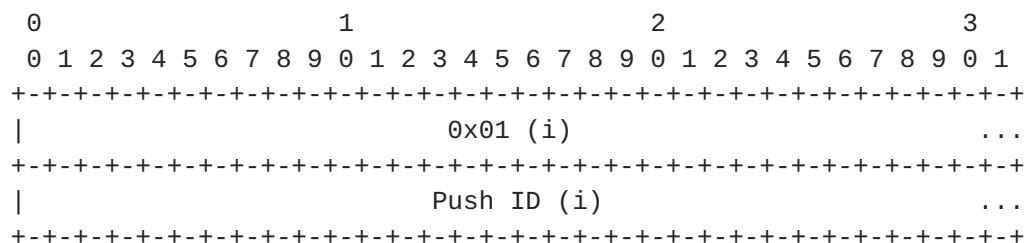
Figure 3: Push Stream Header

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 HTTP_ID_ERROR.

6.2.3. Reserved Stream Types

Stream types of the format "0x1f * N + 0x21" for 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
implementation chooses.

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 the
streams types on which they are permitted; 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
PRIORITY	Yes	No	No	Section 7.2.3
CANCEL_PUSH	Yes	No	No	Section 7.2.4
SETTINGS	Yes (1)	No	No	Section 7.2.5
PUSH_PROMISE	No	Yes	No	Section 7.2.6
GOAWAY	Yes	No	No	Section 7.2.7
MAX_PUSH_ID	Yes	No	No	Section 7.2.8
DUPLICATE_PUSH	No	Yes	No	Section 7.2.9

Table 1: HTTP/3 frames and stream type overview

Certain frames can only occur as the first frame of a particular stream type; these are 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:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Type (i)                           ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Length (i)                          ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      Frame Payload (*)                      ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```
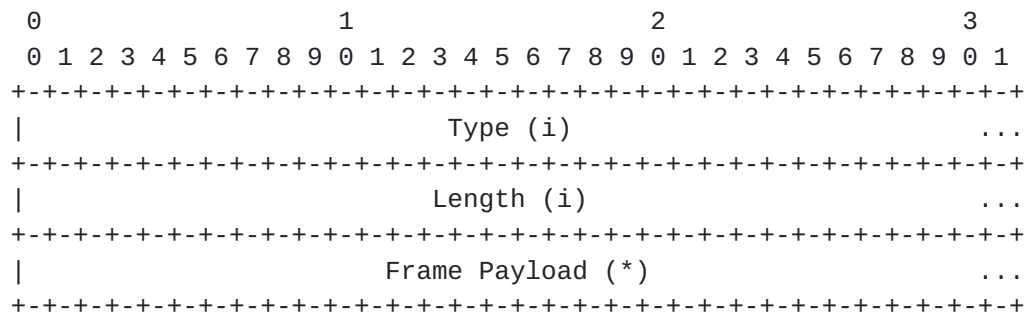
                     Figure 4: HTTP/3 frame format

   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 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 HTTP_MALFORMED_FRAME.

   When a stream terminates cleanly, if the last frame on the stream was
   truncated, this MUST be treated as a connection error (Section 8) of
   type HTTP_MALFORMED_FRAME. Streams which 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 an HTTP request or response payload.

   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 (Section 8) of type
   HTTP_WRONG_STREAM.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Payload (*)                       ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

                    Figure 5: DATA frame payload

7.2.2. HEADERS

   The HEADERS frame (type=0x1) is used to carry a header block,
   compressed using QPACK. See [QPACK] for more details.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Header Block (*)                    ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

                   Figure 6: HEADERS frame payload

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

7.2.3. PRIORITY

   The PRIORITY (type=0x2) frame specifies the client-advised priority
   of a request, server push or placeholder.

   A PRIORITY frame identifies an element to prioritize, and an element
   upon which it depends. A Prioritized ID or Dependency ID identifies
   a client-initiated request using the corresponding stream ID, a
   server push using a Push ID (see Section 7.2.6), or a placeholder
   using a Placeholder ID (see Section 4.3.1).

   In order to ensure that prioritization is processed in a consistent
   order, PRIORITY frames MUST be sent on the control stream.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|PT |DT |X|Empty|          Prioritized Element ID (i)        ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  [Element Dependency ID (i)]               ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Weight (8)  |
+-+-+-+-+-+-+-+-+
```
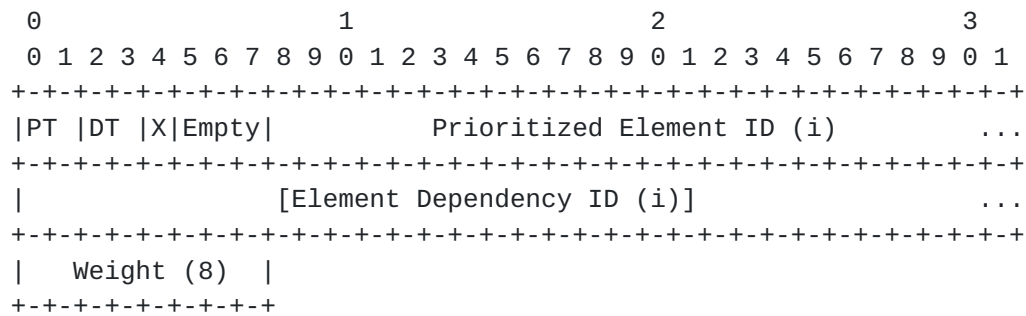
Figure 7: PRIORITY frame payload

The PRIORITY frame payload has the following fields:

PT (Prioritized Element Type): A two-bit field indicating the type
   of element being prioritized (see Table 2). This MUST NOT be set
   to "11".

DT (Element Dependency Type): A two-bit field indicating the type of
   element being depended on (see Table 2).

X (Exclusive Flag): A single-bit flag indicating that the dependency
   is exclusive (see Section 4.3).

Empty: A three-bit field which MUST be zero when sent and has no
   semantic value on receipt.

Prioritized Element ID: A variable-length integer that identifies
   the element being prioritized. Depending on the value of
   Prioritized Type, this contains the Stream ID of a request stream,
   the Push ID of a promised resource, or a Placeholder ID of a
   placeholder.

Element Dependency ID: A variable-length integer that identifies the
   element on which a dependency is being expressed. Depending on
   the value of Dependency Type, this contains the Stream ID of a
   request stream, the Push ID of a promised resource, the
   Placeholder ID of a placeholder, or is absent. For details of
   dependencies, see Section 4.3 and [HTTP2], Section 5.3.

Weight: An unsigned 8-bit integer representing a priority weight for
   the prioritized element (see [HTTP2], Section 5.3). Add one to
   the value to obtain a weight between 1 and 256.

The values for the Prioritized Element Type and Element Dependency
Type (Table 2) imply the interpretation of the associated Element ID
fields.

```
+-----------+-----------------+--------------------+
| Type Bits | Type Description | Element ID Contents |
+-----------+-----------------+--------------------+
| 00        | Request stream  | Stream ID          |
|           |                 |                    |
| 01        | Push stream     | Push ID            |
|           |                 |                    |
| 10        | Placeholder     | Placeholder ID     |
|           |                 |                    |
| 11        | Root of the tree | Absent            |
+-----------+-----------------+--------------------+
```

Table 2: Element Types of a PRIORITY frame

Note that unlike in [HTTP2], the root of the tree cannot be
referenced using a Stream ID of 0, as in QUIC stream 0 carries a
valid HTTP request. The root of the tree cannot be reprioritized.

The PRIORITY frame can express relationships which might not be
permitted based on the stream on which it is sent or its position in
the stream. These situations MUST be treated as a connection error
of type HTTP_MALFORMED_FRAME. The following situations are examples
of invalid PRIORITY frames:

o A PRIORITY frame with the Prioritized Element Type set to "11".

o A PRIORITY frame which claims to reference a request, but the
   associated ID does not identify a client-initiated bidirectional
   stream

A PRIORITY frame with Empty bits not set to zero MAY be treated as a
connection error of type HTTP_MALFORMED_FRAME.

A PRIORITY frame that references a non-existent Push ID, a
Placeholder ID greater than the server's limit, or a Stream ID the
client is not yet permitted to open MUST be treated as a connection
error of type HTTP_ID_ERROR.

A PRIORITY frame received on any stream other than the control stream
MUST be treated as a connection error of type HTTP_WRONG_STREAM.

PRIORITY frames received by a client MUST be treated as a connection
error of type HTTP_UNEXPECTED_FRAME.

7.2.4. **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 7.2.6),
   encoded as a variable-length integer.

   When a server receives this frame, it aborts sending the response for
   the identified server push. If the server has not yet started to
   send the server push, it can use the receipt of a CANCEL_PUSH frame
   to avoid opening a push stream. If the push stream has been opened
   by the server, the server SHOULD send a QUIC RESET_STREAM frame on
   that stream and cease transmission of the response.

   A server can send the CANCEL_PUSH frame to indicate that it will not
   be fulfilling a promise prior to creation of a push stream. Once the
   push stream has been created, sending CANCEL_PUSH has no effect on
   the state of the push stream. A QUIC RESET_STREAM frame SHOULD be
   used instead to abort transmission of the server push response.

   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 HTTP_WRONG_STREAM.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Push ID (i)                        ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```
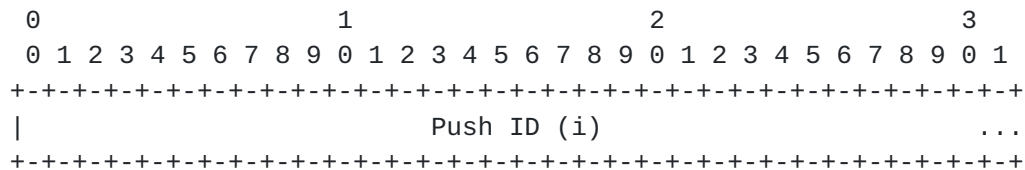
                     Figure 8: CANCEL_PUSH frame payload

   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 7.2.6).

   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.

7.2.5. **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 a 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 HTTP_UNEXPECTED_FRAME.

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
HTTP_WRONG_STREAM.

SETTINGS parameters are not negotiated; they describe characteristics
of the sending peer, which 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 header, while servers are more cautious about request size.

Parameters MUST NOT occur more than once in the SETTINGS frame. A
receiver MAY treat the presence of the same parameter more than once
as a connection error of type HTTP_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.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Identifier (i)                       ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Value (i)                          ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```
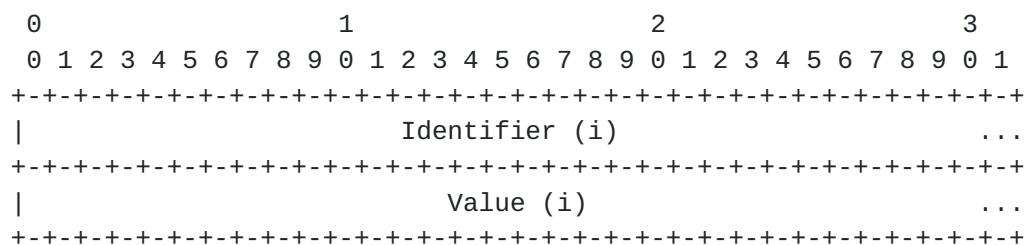
                    Figure 9: SETTINGS parameter format

An implementation MUST ignore the contents for any SETTINGS
identifier it does not understand.

7.2.5.1. Defined SETTINGS Parameters

   The following settings are defined in HTTP/3:

   SETTINGS_MAX_HEADER_LIST_SIZE (0x6): The default value is unlimited.
      See Section 4.1.1 for usage.

   SETTINGS_NUM_PLACEHOLDERS (0x9): The default value is 0. However,
      this value SHOULD be set to a non-zero value by servers. See
      Section 4.3.1 for usage.

   Setting identifiers of the format "0x1f * N + 0x21" for 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.

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

7.2.5.2. Initialization

   An HTTP implementation MUST NOT send frames or requests which would
   be invalid based on its current understanding of the peer's settings.
   All settings begin at an initial value, and are updated upon receipt
   of a SETTINGS frame. 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. 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 MUST store the settings the
   server provided in the session being resumed and MUST comply with
   stored settings until the current server settings are received. A
   client can use these initial values to send requests before the
   server's SETTINGS frame has arrived. This removes the need for a
   client to wait for the SETTINGS frame before sending requests.

   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.

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 MAY
omit settings from its SETTINGS frame which are unchanged from the
initial value.

7.2.6. PUSH_PROMISE

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

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Push ID (i)                        ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Header Block (*)                     ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```
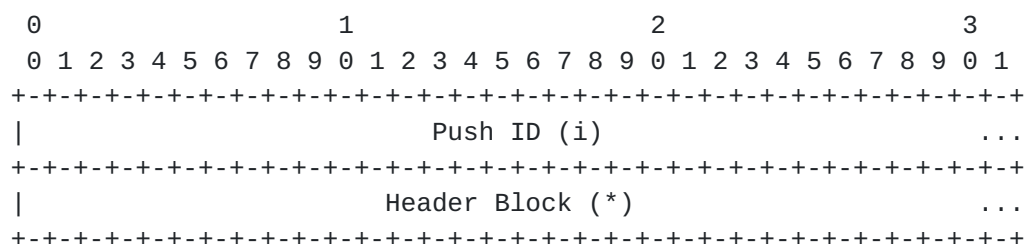
Figure 10: PUSH_PROMISE frame payload

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), CANCEL_PUSH frames (Section 7.2.4), DUPLICATE_PUSH
   frames (Section 7.2.9), and PRIORITY frames (Section 7.2.3).

Header Block: QPACK-compressed 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 7.2.8). 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 HTTP_ID_ERROR.

A server MUST NOT use the same Push ID in multiple PUSH_PROMISE
frames. A client MUST treat receipt of a Push ID which has already
been promised as a connection error of type HTTP_ID_ERROR.

If a PUSH_PROMISE frame is received on the control stream, the client
MUST respond with a connection error (Section 8) of type
HTTP_WRONG_STREAM.

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
HTTP_UNEXPECTED_FRAME.

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

7.2.7. GOAWAY

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

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Stream ID (i)                      ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```
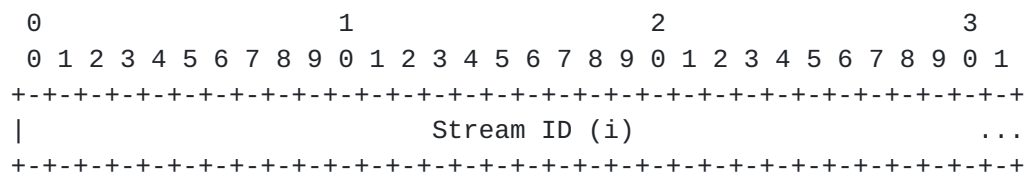
Figure 11: GOAWAY frame payload

The GOAWAY frame is always sent on the control stream. 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 HTTP_MALFORMED_FRAME.

Clients do not need to send GOAWAY to initiate a graceful shutdown;
they simply stop making new requests. A server MUST treat receipt of
a GOAWAY frame on any stream as a connection error (Section 8) of
type HTTP_UNEXPECTED_FRAME.

The GOAWAY frame applies to the connection, not a specific stream. A
client MUST treat a GOAWAY frame on a stream other than the control
stream as a connection error (Section 8) of type HTTP_WRONG_STREAM.

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

7.2.8. 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 a PUSH_PROMISE
frame. Consequently, this also limits the number of push streams
that the server can initiate in addition to the limit set by the QUIC
MAX_STREAMS frame.

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 HTTP_WRONG_STREAM.

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
HTTP_UNEXPECTED_FRAME.

The maximum Push ID is unset when a 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.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Push ID (i)                         ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

                   Figure 12: MAX_PUSH_ID frame payload

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 7.2.6). A MAX_PUSH_ID frame cannot reduce the maximum
Push ID; receipt of a MAX_PUSH_ID that contains a smaller value than
previously received MUST be treated as a connection error of type
HTTP_ID_ERROR.

7.2.9. DUPLICATE_PUSH

The DUPLICATE_PUSH frame (type=0xE) is used by servers to indicate
that an existing pushed resource is related to multiple client
requests.

The DUPLICATE_PUSH frame is always sent on a request stream. Receipt
of a DUPLICATE_PUSH frame on any other stream MUST be treated as a
connection error of type HTTP_WRONG_STREAM.

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

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Push ID (i)                         ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

                   Figure 13: DUPLICATE_PUSH frame payload

The DUPLICATE_PUSH frame carries a single variable-length integer
that identifies the Push ID of a resource that the server has
previously promised (see Section 7.2.6), though that promise might
not be received before this frame. A server MUST NOT use a Push ID
that is larger than the client has provided in a MAX_PUSH_ID frame
(Section 7.2.8). A client MUST treat receipt of a DUPLICATE_PUSH
that contains a larger Push ID than the client has advertised as a
connection error of type HTTP_ID_ERROR.

This frame allows the server to use the same server push in response
to multiple concurrent requests. Referencing the same server push
ensures that a promise can be made in relation to every response in
which server push might be needed without duplicating request headers
or pushed responses.

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 DUPLICATE_PUSH that uses a Push ID that
they have since consumed and discarded are forced to ignore the
DUPLICATE_PUSH.

7.2.10. Reserved Frame Types

Frame types of the format "0x1f * N + 0x21" for integer values of N
are reserved to exercise the requirement that unknown types be
ignored (Section 9). These frames 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 frames to have any meaning upon receipt.

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

8. Error Handling

QUIC allows the application to abruptly terminate (reset) individual
streams or the entire connection when an error is encountered. These
are referred to as "stream errors" or "connection errors" and are
described in more detail in [QUIC-TRANSPORT]. An endpoint MAY choose
to treat a stream error as a connection error.

This section describes HTTP/3-specific error codes which can be used
to express the cause of a connection or stream error.

8.1. HTTP/3 Error Codes

   The following error codes are defined for use in QUIC RESET_STREAM
   frames, STOP_SENDING frames, and CONNECTION_CLOSE frames when using
   HTTP/3.

   HTTP_NO_ERROR (0x00): No error. This is used when the connection or
      stream needs to be closed, but there is no error to signal.

   HTTP_GENERAL_PROTOCOL_ERROR (0x01): Peer violated protocol
      requirements in a way which doesn't match a more specific error
      code, or endpoint declines to use the more specific error code.

   Reserved (0x02): This code is reserved and has no meaning.

   HTTP_INTERNAL_ERROR (0x03): An internal error has occurred in the
      HTTP stack.

   Reserved (0x04): This code is reserved and has no meaning.

   HTTP_REQUEST_CANCELLED (0x05): The request or its response
      (including pushed response) is cancelled.

   HTTP_INCOMPLETE_REQUEST (0x06): The client's stream terminated
      without containing a fully-formed request.

   HTTP_CONNECT_ERROR (0x07): The connection established in response to
      a CONNECT request was reset or abnormally closed.

   HTTP_EXCESSIVE_LOAD (0x08): The endpoint detected that its peer is
      exhibiting a behavior that might be generating excessive load.

   HTTP_VERSION_FALLBACK (0x09): The requested operation cannot be
      served over HTTP/3. The peer should retry over HTTP/1.1.

   HTTP_WRONG_STREAM (0x0A): A frame was received on a stream where it
      is not permitted.

   HTTP_ID_ERROR (0x0B): A Stream ID, Push ID, or Placeholder ID was
      used incorrectly, such as exceeding a limit, reducing a limit, or
      being reused.

   Reserved (0x0C): N/A

   HTTP_STREAM_CREATION_ERROR (0x0D): The endpoint detected that its
      peer created a stream that it will not accept.

   Reserved (0x0E): N/A

   HTTP_CLOSED_CRITICAL_STREAM (0x0F): A stream required by the
      connection was closed or reset.

   Reserved (0x0010): N/A

   HTTP_EARLY_RESPONSE (0x0011): The remainder of the client's request
      is not needed to produce a response. For use in STOP_SENDING
      only.

   HTTP_MISSING_SETTINGS (0x0012): No SETTINGS frame was received at
      the beginning of the control stream.

   HTTP_UNEXPECTED_FRAME (0x0013): A frame was received which was not
      permitted in the current state.

   HTTP_REQUEST_REJECTED (0x0014): A server rejected a request without
      performing any application processing.

   HTTP_SETTINGS_ERROR (0x00FF): An endpoint detected an error in the
      payload of a SETTINGS frame: a duplicate setting was detected, a
      client-only setting was sent by a server, or a server-only setting
      by a client.

   HTTP_MALFORMED_FRAME (0x01XX): An error in a specific frame type.
      If the frame type is "0xfe" or less, the type is included as the
      last byte of the error code. For example, an error in a
      MAX_PUSH_ID frame would be indicated with the code (0x10D). The
      last byte "0xff" is used to indicate any frame type greater than
      "0xfe".

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 header fields.

   Extensions are permitted to use new frame types (Section 7.2), new
   settings (Section 7.2.5.1), new error codes (Section 8), or new
   unidirectional stream types (Section 6.2). Registries are
   established for managing these extension points: frame types
   (Section 11.3), settings (Section 11.4), error codes (Section 11.5),
   and stream types (Section 11.6).

Implementations MUST ignore unknown or unsupported values in all
extensible protocol elements. Implementations MUST discard frames
and 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.

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.
In this case, it could also be necessary to coordinate when the
revised layout comes into effect.

This document doesn't mandate a specific method for negotiating the
use of an extension but notes that a setting (Section 7.2.5.1) 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. Note that where HTTP/2 employs PADDING frames
and Padding fields in other frames to make a connection more
resistant to traffic analysis, HTTP/3 can rely on QUIC PADDING frames
or employ the reserved frame and stream types discussed in
Section 7.2.10 and Section 6.2.3.

When HTTP Alternative Services is used for discovery for HTTP/3
endpoints, the security considerations of [ALTSVC] also apply.

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.

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.

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.

11. IANA Considerations

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. Registration of QUIC Version Hint Alt-Svc Parameter

This document creates a new registration for version-negotiation
hints in the "Hypertext Transfer Protocol (HTTP) Alt-Svc Parameter"
registry established in [RFC7838].

Parameter: "quic"

Specification: This document, Section 3.2.1

11.3. 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 space
is split into three spaces that are governed by different policies.
Values between "0x00" and "0x3f" (in hexadecimal) are assigned via
the Standards Action or IESG Review policies [RFC8126]. Values from
"0x40" to "0x3fff" operate on the Specification Required policy
[RFC8126]. All other values are assigned to Private Use [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.

New entries in this registry require the following information:

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

   Code: The 62-bit code assigned to the frame type.

   Specification: A reference to a specification that includes a
      description of the frame layout and its semantics, including any
      parts of the frame that are conditionally present.

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

```
         +-----------------+------+---------------+
         | Frame Type      | Code | Specification |
         +-----------------+------+---------------+
         | DATA            | 0x0  | Section 7.2.1 |
         |                 |      |               |
         | HEADERS         | 0x1  | Section 7.2.2 |
         |                 |      |               |
         | PRIORITY        | 0x2  | Section 7.2.3 |
         |                 |      |               |
         | CANCEL_PUSH     | 0x3  | Section 7.2.4 |
         |                 |      |               |
         | SETTINGS        | 0x4  | Section 7.2.5 |
         |                 |      |               |
         | PUSH_PROMISE    | 0x5  | Section 7.2.6 |
         |                 |      |               |
         | Reserved        | 0x6  | N/A           |
         |                 |      |               |
         | GOAWAY          | 0x7  | Section 7.2.7 |
         |                 |      |               |
         | Reserved        | 0x8  | N/A           |
         |                 |      |               |
         | Reserved        | 0x9  | N/A           |
         |                 |      |               |
         | MAX_PUSH_ID     | 0xD  | Section 7.2.8 |
         |                 |      |               |
         | DUPLICATE_PUSH  | 0xE  | Section 7.2.9 |
         +-----------------+------+---------------+
```

   Additionally, each code of the format "0x1f * N + 0x21" for integer
   values of N (that is, "0x21", "0x40", ..., through
   "0x3FFFFFFFFFFFFFFE") MUST NOT be assigned by IANA.

11.4. Settings Parameters

   This document establishes a registry for HTTP/3 settings. The
   "HTTP/3 Settings" registry governs a 62-bit space. This space is
   split into three spaces that are governed by different policies.
   Values between "0x00" and "0x3f" (in hexadecimal) are assigned via
   the Standards Action or IESG Review policies [RFC8126]. Values from
   "0x40" to "0x3fff" operate on the Specification Required policy

[RFC8126]. All other values are assigned to Private Use [RFC8126].
The designated experts are the same as those for the "HTTP/2
Settings" registry defined in [HTTP2].

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.

New registrations are advised to provide the following information:

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

Code: The 62-bit code assigned to the setting.

Specification: An optional reference to a specification that
   describes the use of the setting.

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

```
        +----------------------+------+-----------------+
        | Setting Name         | Code | Specification   |
        +----------------------+------+-----------------+
        | Reserved             | 0x2  | N/A             |
        |                      |      |                 |
        | Reserved             | 0x3  | N/A             |
        |                      |      |                 |
        | Reserved             | 0x4  | N/A             |
        |                      |      |                 |
        | Reserved             | 0x5  | N/A             |
        |                      |      |                 |
        | MAX_HEADER_LIST_SIZE | 0x6  | Section 7.2.5.1 |
        |                      |      |                 |
        | NUM_PLACEHOLDERS     | 0x9  | Section 7.2.5.1 |
        +----------------------+------+-----------------+
```

Additionally, each code of the format "0x1f * N + 0x21" for integer
values of N (that is, "0x21", "0x40", ..., through
"0x3FFFFFFFFFFFFFFE") MUST NOT be assigned by IANA.

11.5. Error Codes

This document establishes a registry for HTTP/3 error codes. The
"HTTP/3 Error Code" registry manages a 62-bit space. The "HTTP/3
Error Code" registry operates under the "Expert Review" policy
[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.

New registrations are advised to provide the following information:

Name: A name for the error code. Specifying an error code name is
   optional.

Code: The 62-bit error code value.

Description: A brief description of the error code semantics, longer
   if no detailed specification is provided.

Specification: An optional reference for a specification that
   defines the error code.

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

Name	Code	Description	Specification
HTTP_NO_ERROR	0x0000	No error	Section 8.1
HTTP_GENERAL_PROTOCOL_ERRO R	0x0001	General protocol error	Section 8.1
Reserved	0x0002	N/A	N/A
HTTP_INTERNAL_ERROR	0x0003	Internal error	Section 8.1
Reserved	0x0004	N/A	N/A
HTTP_REQUEST_CANCELLED	0x0005	Data no longer needed	Section 8.1
HTTP_INCOMPLETE_REQUEST	0x0006	Stream terminated early	Section 8.1
HTTP_CONNECT_ERROR	0x0007	TCP reset or error on CONNECT request	Section 8.1

HTTP_EXCESSIVE_LOAD	0x0008	Peer generating excessive load	Section 8.1	
HTTP_VERSION_FALLBACK	0x0009	Retry over HTTP/1.1	Section 8.1	
HTTP_WRONG_STREAM	0x000A	A frame was sent on the wrong stream	Section 8.1	
HTTP_ID_ERROR	0x000B	An identifier was used incorrectly	Section 8.1	
Reserved	0x000C	N/A	N/A	
HTTP_STREAM_CREATION_ERROR	0x000D	Stream creation error	Section 8.1	
Reserved	0x000E	N/A	N/A	
HTTP_CLOSED_CRITICAL_STREAM	0x000F	Critical stream was closed	Section 8.1	
Reserved	0x000E	N/A	N/A	
HTTP_EARLY_RESPONSE	0x0011	Remainder of request not needed	Section 8.1	
HTTP_MISSING_SETTINGS	0x0012	No SETTINGS frame received	Section 8.1	
HTTP_UNEXPECTED_FRAME	0x0013	Frame not permitted in the current state	Section 8.1	
HTTP_REQUEST_REJECTED	0x0014	Request not	Section 8.1	

```
| |          |          | processed  |          |
| |          |          |            |          |
| HTTP_MALFORMED_FRAME   | 0x01XX | Error in | Section 8.1 |
| |          |          | frame      |          |
| |          |          | formatting |          |
| |          |          |            |          |
| HTTP_SETTINGS_ERROR    | 0x00FF | SETTINGS | Section 8.1 |
| |          |          | frame      |          |
| |          |          | contained  |          |
| |          |          | invalid    |          |
| |          |          | values     |          |
+---------------------------+--------+------------+--------------+
```

11.6. 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 space is split into three spaces that are governed by different
   policies. Values between "0x00" and 0x3f (in hexadecimal) are
   assigned via the Standards Action or IESG Review policies [RFC8126].
   Values from "0x40" to "0x3fff" operate on the Specification Required
   policy [RFC8126]. All other values are assigned to Private Use
   [RFC8126].

   New entries in this registry require the following information:

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

   Code: The 62-bit code assigned to the stream type.

   Specification: A reference to a specification that includes a
      description of the stream type, including the layout semantics of
      its payload.

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

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

```
+----------------+------+---------------+--------+
| Stream Type    | Code | Specification | Sender |
+----------------+------+---------------+--------+
| Control Stream | 0x00 | Section 6.2.1 | Both   |
|                |      |               |        |
| Push Stream    | 0x01 | Section 4.4   | Server |
+----------------+------+---------------+--------+
```

Additionally, each code of the format "0x1f * N + 0x21" for integer
values of N (that is, "0x21", "0x40", ..., through
"0x3FFFFFFFFFFFFFFE") MUST NOT be assigned by IANA.

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>.

[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>.

[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>.

[QPACK]   Krasic, C., Bishop, M., and A. Frindell, Ed., "QPACK:
            Header Compression for HTTP over QUIC", draft-ietf-quic-
            qpack-10 (work in progress), July 2019.

[QUIC-TRANSPORT]
            Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based
            Multiplexed and Secure Transport", draft-ietf-quic-
            transport-22 (work in progress), July 2019.

[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>.

[RFC5234]   Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax
            Specifications: ABNF", STD 68, RFC 5234,
            DOI 10.17487/RFC5234, January 2008,
            <https://www.rfc-editor.org/info/rfc5234>.

   [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>.

   [RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer
              Protocol (HTTP/1.1): Message Syntax and Routing",
              RFC 7230, DOI 10.17487/RFC7230, June 2014,
              <https://www.rfc-editor.org/info/rfc7230>.

   [RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer
              Protocol (HTTP/1.1): Semantics and Content", RFC 7231,
              DOI 10.17487/RFC7231, June 2014,
              <https://www.rfc-editor.org/info/rfc7231>.

   [RFC7540] 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>.

   [RFC7838] 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>.

   [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>.

12.2. Informative References

   [HPACK]   Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer
              Protocol (HTTP/1.1): Semantics and Content", RFC 7231,
              DOI 10.17487/RFC7231, June 2014,
              <https://www.rfc-editor.org/info/rfc7231>.

   [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>.

   [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>.

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

12.3. URIs

   [1] https://mailarchive.ietf.org/arch/search/?email_list=quic

   [2] https://github.com/quicwg

   [3] https://github.com/quicwg/base-drafts/labels/-http

   [4] https://www.iana.org/assignments/message-headers

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 than HTTP/2.

   These 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 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.

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) which 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 equivalent frames between the
two mappings are not identical.

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. Implicit in the HTTP/2
prioritization scheme is the notion of in-order delivery of priority
changes (i.e., dependency tree mutations). Since operations on the
dependency tree such as reparenting a subtree are not commutative,
both sender and receiver must apply them in the same order to ensure
that both sides have a consistent view of the stream dependency tree.

To achieve in-order delivery of priority changes in HTTP/3, PRIORITY
frames are sent on the control stream. HTTP/3 permits the
prioritization of requests, pushes and placeholders that each exist
in separate identifier spaces. The HTTP/3 PRIORITY frame replaces
the stream dependency field with fields that can identify the element
of interest and its dependency.

A.2.2. Header Compression Differences

HPACK was designed with the assumption of in-order delivery. A
sequence of encoded header blocks 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 which contain encoded
headers merely reference the table state at a given time without
modifying it.

[QPACK] provides additional details.

A.2.3. 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 rather than a
Stream ID (e.g. Push IDs in PRIORITY frames). 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 which depend on the presence of flags need to allocate
space for flags as part of their frame payload.

Other than this issue, 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 would be portable
to HTTP/2 in the same manner.

A.2.4. Mapping 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. A separate PRIORITY frame is used in all cases.
   Padding is not defined in HTTP/3 frames. See Section 7.2.2.

PRIORITY (0x2): As described above, the PRIORITY frame references a
   variety of identifiers. It is sent as the first frame on a
   request streams or on the control stream. See Section 7.2.3.

RST_STREAM (0x3): RST_STREAM frames do not exist, since QUIC
   provides stream lifecycle management. The same code point is used
   for the CANCEL_PUSH frame (Section 7.2.4).

SETTINGS (0x4): SETTINGS frames are sent only at the beginning of
   the connection. See Section 7.2.5 and Appendix A.3.

   PUSH_PROMISE (0x5): The PUSH_PROMISE does not reference a stream;
      instead the push stream references the PUSH_PROMISE frame using a
      Push ID. See Section 7.2.6.

   PING (0x6): PING frames do not exist, since QUIC provides equivalent
      functionality.

   GOAWAY (0x7): GOAWAY is sent only from server to client and does not
      contain an error code. See Section 7.2.7.

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

   CONTINUATION (0x9): CONTINUATION frames do not exist; 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 11.3.

A.3. HTTP/2 SETTINGS Parameters

   An important difference from HTTP/2 is that settings are sent once,
   at the beginning of the connection, 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 options that are retained in HTTP/3 have the same value as
   in HTTP/2.

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

   SETTINGS_HEADER_TABLE_SIZE: See [QPACK].

   SETTINGS_ENABLE_PUSH: This is removed in favor of the MAX_PUSH_ID
      which provides a more granular control over server push.

   SETTINGS_MAX_CONCURRENT_STREAMS: QUIC controls the largest open
      Stream ID as part of its flow control logic. Specifying
      SETTINGS_MAX_CONCURRENT_STREAMS in the SETTINGS frame is an error.

   SETTINGS_INITIAL_WINDOW_SIZE: QUIC requires both stream and
      connection flow control window sizes to be specified in the

      initial transport handshake. Specifying
      SETTINGS_INITIAL_WINDOW_SIZE in the SETTINGS frame is an error.

   SETTINGS_MAX_FRAME_SIZE: This setting has no equivalent in HTTP/3.
      Specifying it in the SETTINGS frame is an error.

   SETTINGS_MAX_HEADER_LIST_SIZE: See Section 7.2.5.1.

   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 which use the full 32-bit space. Settings
   ported from HTTP/2 might choose to redefine the format of their
   settings to avoid using the 62-bit encoding.

   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 11.4.

A.4. HTTP/2 Error Codes

   QUIC has the same concepts of "stream" and "connection" errors that
   HTTP/2 provides. However, there is no direct portability of HTTP/2
   error codes.

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

   NO_ERROR (0x0): HTTP_NO_ERROR in Section 8.1.

   PROTOCOL_ERROR (0x1): This is mapped to HTTP_GENERAL_PROTOCOL_ERROR
      except in cases where more specific error codes have been defined.
      This includes HTTP_MALFORMED_FRAME, HTTP_WRONG_STREAM,
      HTTP_UNEXPECTED_FRAME and HTTP_CLOSED_CRITICAL_STREAM defined in
      Section 8.1.

   INTERNAL_ERROR (0x2): HTTP_INTERNAL_ERROR in Section 8.1.

   FLOW_CONTROL_ERROR (0x3): Not applicable, since QUIC handles flow
      control. Would provoke a QUIC_FLOW_CONTROL_RECEIVED_TOO_MUCH_DATA
      from the QUIC layer.

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

STREAM_CLOSED (0x5): Not applicable, since QUIC handles stream
   management. Would provoke a QUIC_STREAM_DATA_AFTER_TERMINATION
   from the QUIC layer.

FRAME_SIZE_ERROR (0x6): HTTP_MALFORMED_FRAME error codes defined in
   Section 8.1.

REFUSED_STREAM (0x7): HTTP_REQUEST_REJECTED (in Section 8.1) is used
   to indicate that a request was not processed. Otherwise, not
   applicable because QUIC handles stream management. A
   STREAM_ID_ERROR at the QUIC layer is used for streams that are
   improperly opened.

CANCEL (0x8): HTTP_REQUEST_CANCELLED in Section 8.1.

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

CONNECT_ERROR (0xa): HTTP_CONNECT_ERROR in Section 8.1.

ENHANCE_YOUR_CALM (0xb): HTTP_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): HTTP_VERSION_FALLBACK in Section 8.1.

Error codes need to be defined for HTTP/2 and HTTP/3 separately. See
Section 11.5.

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-21

o No changes

B.2. Since draft-ietf-quic-http-20

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

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

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

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

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

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

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

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

o 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.3. **Since draft-ietf-quic-http-19**

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

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

B.4. **Since draft-ietf-quic-http-18**

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

o 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

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

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

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

B.5. Since draft-ietf-quic-http-17

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

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

B.6. Since draft-ietf-quic-http-16

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

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

   * Permitted as first frame of request streams

   * Remove exclusive reprioritization

   * Changes to Prioritized Element Type bits

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

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

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

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

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

B.7. Since **draft-ietf-quic-http-15**

   Substantial editorial reorganization; no technical changes.

B.8. Since **draft-ietf-quic-http-14**

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

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

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

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

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

B.9. Since **draft-ietf-quic-http-13**

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

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

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

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

B.10. Since **draft-ietf-quic-http-12**

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

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

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

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

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

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

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

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

B.12. Since **draft-ietf-quic-http-10**

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

B.13. Since **draft-ietf-quic-http-09**

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

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

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

B.14. Since **draft-ietf-quic-http-08**

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

B.15. Since **draft-ietf-quic-http-07**

   o Changes for integer encodings in QUIC (#595,#905)

   o Use unidirectional streams as appropriate (#515, #240, #281, #886)

   o Improvement to the description of GOAWAY (#604, #898)

   o Improve description of server push usage (#947, #950, #957)

B.16. Since **draft-ietf-quic-http-06**

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

B.17. Since **draft-ietf-quic-http-05**

   o Made push ID sequential, add MAX_PUSH_ID, remove
      SETTINGS_ENABLE_PUSH (#709)

   o Guidance about keep-alive and QUIC PINGs (#729)

   o Expanded text on GOAWAY and cancellation (#757)

B.18. Since **draft-ietf-quic-http-04**

   o Cite RFC 5234 (#404)

   o Return to a single stream per request (#245,#557)

   o Use separate frame type and settings registries from HTTP/2 (#81)

   o SETTINGS_ENABLE_PUSH instead of SETTINGS_DISABLE_PUSH (#477)

   o Restored GOAWAY (#696)

   o Identify server push using Push ID rather than a stream ID
      (#702,#281)

   o DATA frames cannot be empty (#700)

B.19. Since **draft-ietf-quic-http-03**

   None.

B.20. Since **draft-ietf-quic-http-02**

   o Track changes in transport draft

B.21. Since **draft-ietf-quic-http-01**

   o 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

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

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

   o HPACK Sequence counter can wrap (#173)

   o 0-RTT guidance added

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

B.22. **Since draft-ietf-quic-http-00**

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

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

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

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

   o Described CONNECT pseudo-method (#95)

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

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

B.23. **Since draft-shade-quic-http2-mapping-00**

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

   o Updated authors/editors list

Acknowledgements

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

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

Author's Address

   Mike Bishop (editor)
   Akamai

   Email: mbishop@evequefou.be