Hypertext Transfer Protocol (HTTP) over QUIC
draft-ietf-quic-http-15

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 QUIC.

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].

Copyright Notice

Table of Contents

1. Introduction

   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, drawing heavily on
   the existing TCP mapping, HTTP/2. Specifically, this document
   identifies HTTP/2 features that are subsumed by QUIC, and describes
   how the other features can be implemented atop QUIC.

   QUIC is described in [QUIC-TRANSPORT]. For a full description of
   HTTP/2, see [RFC7540].

1.1. Notational Conventions

   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].

   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
   APPLICATION_CLOSE frames." References without this preface refer to
   frames defined in Section 4.2.

2. Connection Setup and Management

2.1. Draft Version Identification

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

   HTTP/QUIC uses the token "hq" to identify itself in ALPN and Alt-Svc.
   Only implementations of the final, published RFC can identify
   themselves as "hq". 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 "hq-
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 "hq-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.

2.2. Discovering an HTTP/QUIC Endpoint

An HTTP origin advertises the availability of an equivalent HTTP/QUIC
endpoint via the Alt-Svc HTTP response header field or the HTTP/2
ALTSVC frame ([ALTSVC]), using the ALPN token defined in Section 2.3.

For example, an origin could indicate in an HTTP/1.1 or HTTP/2
response that HTTP/QUIC was available on UDP port 50781 at the same
hostname by including the following header field in any response:

Alt-Svc: hq=":50781"

On receipt of an Alt-Svc record indicating HTTP/QUIC 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/QUIC on any UDP port, since an alternative
always includes an explicit port.

2.2.1. QUIC Version Hints

This document defines the "quic" parameter for Alt-Svc, which MAY be
used to provide version-negotiation hints to HTTP/QUIC clients. QUIC
versions are four-octet 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 opted to include the
reserved versions (from Section 4 of [QUIC-TRANSPORT]) 0x0 and
0x1abadaba, it could specify the following header field:

Alt-Svc: hq=":49288";quic="1,1abadaba,51303334,0"

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

2.3. Connection Establishment

HTTP/QUIC 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/QUIC 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/QUIC support is indicated by
selecting the ALPN token "hq" 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/QUIC-specific settings
are conveyed in the SETTINGS frame. After the QUIC connection is
established, a SETTINGS frame (Section 4.2.6) MUST be sent by each
endpoint as the initial frame of their respective HTTP control stream
(see Section 3.3.2). The server MUST NOT send data on any other
stream until the client's SETTINGS frame has been received.

2.4. 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/QUIC 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/QUIC 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 [RFC7540]).

The considerations discussed in Section 9.1 of [RFC7540] also apply
to the management of HTTP/QUIC connections.

3. 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. A QUIC receiver
buffers and orders received STREAM frames, exposing the data
contained within as a reliable byte stream to the application.

When HTTP headers and data are sent over QUIC, the QUIC layer handles
most of the stream management.

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/QUIC client SHOULD send non-zero values for the QUIC transport
parameters "initial_max_stream_data_bidi_local". An HTTP/QUIC 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.

These streams carry frames related to the request/response (see
Section 4.2). When a stream terminates cleanly, if the last frame on
the stream was truncated, this MUST be treated as a connection error
(see HTTP_MALFORMED_FRAME in Section 6.1). Streams which terminate
abruptly may be reset at any point in the frame.

HTTP/QUIC does not use server-initiated bidirectional streams. The
use of unidirectional streams is discussed in Section 3.3. Both
clients and servers SHOULD send a value of three or greater for the
QUIC transport parameter "initial_max_uni_streams".

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. Requests and responses are considered complete when the
corresponding QUIC stream is closed in the appropriate direction.

3.1. HTTP Message Exchanges

A client sends an HTTP request on a client-initiated bidirectional
QUIC stream. A server sends an HTTP response on the same stream as
the request.

An HTTP message (request or response) consists of:

1. one header block (see Section 4.2.3) containing the message
    header (see [RFC7230], Section 3.2),

2. the payload body (see [RFC7230], Section 3.3), sent as a series
    of DATA frames (see Section 4.2.2),

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

In addition, prior to sending the message header block indicated
above, a response may contain zero or more header blocks containing
the message headers of informational (1xx) HTTP responses (see
[RFC7230], Section 3.2 and [RFC7231], Section 6.2).

A server MAY interleave one or more PUSH_PROMISE frames (see
Section 4.2.7) with the frames of a response message. These
PUSH_PROMISE frames are not part of the response; see Section 3.3.3
for more details.

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

Trailing header fields are carried in an additional header block
following the body. Senders MUST send only one header block in the

trailers section; receivers MUST discard any subsequent header
blocks.

An HTTP request/response exchange fully consumes a bidirectional QUIC
stream. After sending a request, a client closes the stream for
sending; after sending a response, the server closes the stream for
sending and the QUIC stream is fully closed.

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 with error code
HTTP_EARLY_RESPONSE, sending a complete response, and cleanly closing
its streams. 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.

Changes to the state of a request stream, including receiving a
RST_STREAM with any error code, do not affect the state of the
server's response. Servers do not abort a response in progress
solely due to a state change on the request stream. However, if the
request stream terminates without containing a usable HTTP request,
the server SHOULD abort its response with the error code
HTTP_INCOMPLETE_REQUEST.

3.1.1. Header Formatting and Compression

HTTP header fields carry information as a series of key-value pairs.
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/QUIC 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.

As in HTTP/2, HTTP/QUIC uses special pseudo-header fields beginning
with ':' 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
[RFC7540]. Pseudo-header fields are not HTTP header fields.
Endpoints MUST NOT generate pseudo-header fields other than those

   defined in [RFC7540]. The restrictions on the use of pseudo-header
   fields in Section 8.1.2.1 of [RFC7540] also apply to HTTP/QUIC.

   HTTP/QUIC 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.

3.1.2. 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/QUIC functions in the same manner as in
   HTTP/2. The request MUST be formatted as described in [RFC7540],
   Section 8.3. A CONNECT request that does not conform to these
   restrictions is malformed. The request stream MUST NOT be half-
   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 request stream correspond to data sent 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.

   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
   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 RST_STREAM. A proxy treats
any error in the TCP connection, which includes receiving a TCP
segment with the RST bit set, as a stream error of type
HTTP_CONNECT_ERROR (Section 6.1). Correspondingly, a proxy MUST send
a TCP segment with the RST bit set if it detects an error with the
stream or the QUIC connection.

3.1.3. Request Cancellation

Either client or server can cancel requests by aborting the stream
(QUIC RST_STREAM or STOP_SENDING frames, as appropriate) with an
error code of HTTP_REQUEST_CANCELLED (Section 6.1). When the client
cancels a response, it indicates that this response is no longer of
interest. Clients SHOULD cancel requests by aborting both directions
of a stream.

When the server cancels its response stream using
HTTP_REQUEST_CANCELLED, it indicates that no application processing
was performed. The client can treat requests cancelled 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_CANCELLED status for requests which were partially or
fully processed.

Note: 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.

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).

3.2. Request Prioritization

HTTP/QUIC uses a priority scheme similar to that described in
[RFC7540], Section 5.3. In this priority scheme, a given stream can
be designated as dependent upon another request, which expresses the
preference that the latter stream (the "parent" request) be allocated
resources before the former stream (the "dependent" request). Taken
together, the dependencies across all requests in a connection form a
dependency tree. The structure of the dependency tree changes as
PRIORITY frames add, remove, or change the dependency links between
requests.

The PRIORITY frame Section 4.2.4 identifies a prioritized element.
The elements which 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 4.2.7)

o Placeholders, identified by a Placeholder ID

An element can depend on another element or on the root of the tree.
A reference to an element which is no longer in the tree is treated
as a reference to the root of the tree.

Only a client can send PRIORITY frames. A server MUST NOT send a
PRIORITY frame.

3.2.1. Placeholders

In HTTP/2, certain implementations used closed or unused streams as
placeholders in describing the relative priority of requests.
However, this created confusion as servers could not reliably
identify which elements of the priority tree could safely be
discarded. 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/QUIC, a number of placeholders are explicitly permitted by
the server using the "SETTINGS_NUM_PLACEHOLDERS" setting. Because
the server commits to maintain these IDs in the tree, clients can use
them with confidence that the server will not have discarded the
state.

Placeholders are identified by an ID between zero and one less than
the number of placeholders the server has permitted.

3.2.2. Priority Tree Maintenance

Servers can aggressively prune inactive regions from the priority
tree, because placeholders will be used to "root" any persistent
structure of the tree which the client cares about retaining. 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
```
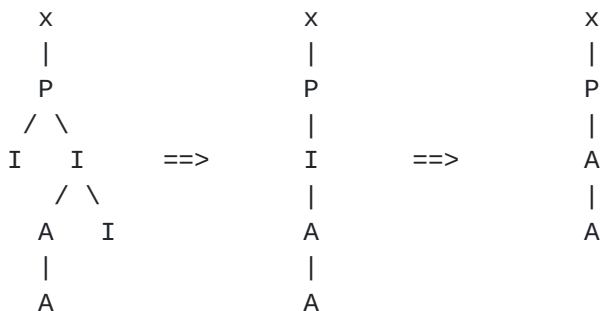
                  Figure 1: Example of Priority Tree Pruning

   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.

3.3. 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 single octet header at the start of the stream. The format and
   structure of data that follows this header is determined by the
   stream type.

```
 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+
|Stream Type (8)|
+-+-+-+-+-+-+-+-+
```

                   Figure 2: Unidirectional Stream Header

   Some stream types are reserved (Section 3.3.1). Two stream types are
   defined in this document: control streams (Section 3.3.2) and push
   streams (Section 3.3.3). Other stream types can be defined by
   extensions to HTTP/QUIC.

   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_UNKNOWN_STREAM_TYPE, but MUST NOT consider such streams to be an
   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.

3.3.1. Reserved Stream Types

   Stream types of the format "0x1f * N" are reserved to exercise the
   requirement that unknown types be ignored. These streams have no
   semantic meaning, and can be sent when application-layer padding is
   desired. They MAY also be sent on connections where no request 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.

3.3.2. Control Streams

   The control stream is indicated by a stream type of "0x43" (ASCII
   'C'). Data on this stream consists of HTTP/QUIC frames, as defined
   in Section 4.2.

   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_WRONG_STREAM_COUNT. If the 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

they are 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.

3.3.3. Server Push

HTTP/QUIC server push is similar to what is described in HTTP/2
[RFC7540], but uses different mechanisms.

The PUSH_PROMISE frame (Section 4.2.7) is sent on the client-
initiated bidirectional stream that carried the request that
generated the push. This allows the server push to be associated
with a request. Ordering of a PUSH_PROMISE in relation to certain
parts of the response is important (see Section 8.2.1 of [RFC7540]).

The PUSH_PROMISE frame does not reference a stream; it contains a
Push ID that uniquely identifies a server push. This allows a server
to fulfill promises in the order that best suits its needs. The same
Push ID can be used in multiple PUSH_PROMISE frames (see
Section 4.2.7). When a server later fulfills a promise, the server
push response is conveyed on a push stream.

A push stream is indicated by a stream type of "0x50" (ASCII 'P'),
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/QUIC frames, as defined in Section 4.2, and carries the
response side of an HTTP message exchange as described in
Section 3.1. The header of the request message is carried by a
PUSH_PROMISE frame (see Section 4.2.7) on the request stream which
generated the push. Promised requests MUST conform to the
requirements in Section 8.2 of [RFC7540].

Only servers can push; if a server receives a client-initiated push
stream, this MUST be treated as a stream error of type
HTTP_WRONG_STREAM_DIRECTION.

```
 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 (8)|               Push ID (i)                 ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```
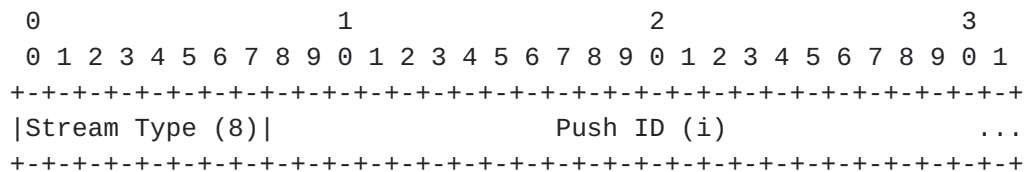
Figure 3: Push Stream Header

Server push is only enabled on a connection when a client sends a
MAX_PUSH_ID frame (see Section 4.2.9). 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_PUSH_LIMIT_EXCEEDED.

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_DUPLICATE_PUSH.

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,
a QUIC STOP_SENDING frame with an appropriate error code can be used
instead (e.g., HTTP_PUSH_REFUSED, HTTP_PUSH_ALREADY_IN_CACHE; see
Section 6). This asks the server not to transfer the data and
indicates that it will be discarded upon receipt.

4. HTTP Framing Layer

Frames are used on the control stream, request streams, and push
streams. This section describes HTTP framing in QUIC and highlights
some differences from HTTP/2 framing. For more detail on differences
from HTTP/2, see Section 8.2.

4.1. 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Length (i)                          ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Type (8)    |            Frame Payload (*)                ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```
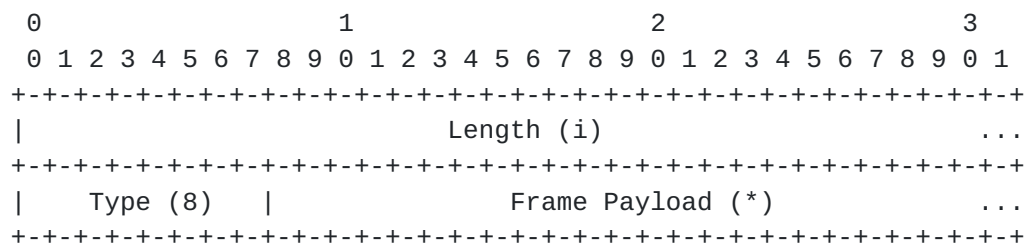
Figure 4: HTTP/QUIC frame format

A frame includes the following fields:

Length: A variable-length integer that describes the length of the
   Frame Payload. This length does not include the Type field.

Type: An 8-bit type for the frame.

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

4.2. Frame Definitions

4.2.1. Reserved Frame Types

Frame types of the format "0xb + (0x1f * N)" are reserved to exercise
the requirement that unknown types be ignored. These frames have no
semantic meaning, and can be sent when application-layer padding is
desired. They MAY also be sent on connections where no request 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.

4.2.2. DATA

DATA frames (type=0x0) convey arbitrary, variable-length sequences of
octets 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 either control stream, the recipient MUST
respond with a connection error (Section 6) 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 (*)                         ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```
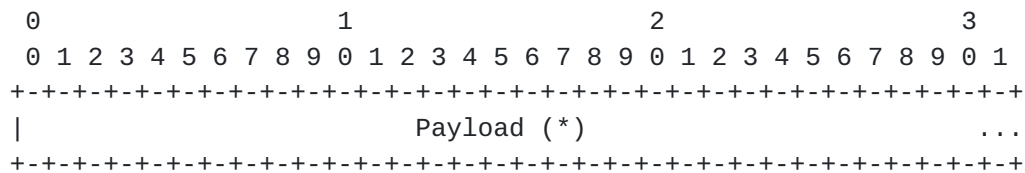
Figure 5: DATA frame payload

DATA frames MUST contain a non-zero-length payload. If a DATA frame
is received with a payload length of zero, the recipient MUST respond
with a stream error (Section 6) of type HTTP_MALFORMED_FRAME.

4.2.3. 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 (*)                      ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```
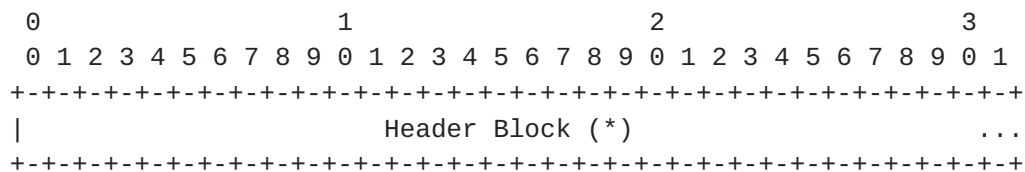
Figure 6: HEADERS frame payload

HEADERS frames can only be sent on request / push streams.

4.2.4. PRIORITY

The PRIORITY (type=0x02) frame specifies the sender-advised priority
of a stream and is substantially different in format from [RFC7540].
In order to ensure that prioritization is processed in a consistent
order, PRIORITY frames MUST be sent on the control stream. A
PRIORITY frame sent on any other stream MUST be treated as a
HTTP_WRONG_STREAM error.

The format has been modified to accommodate not being sent on a
request stream, to allow for identification of server pushes, and the
larger stream ID space of QUIC. The semantics of the Stream
Dependency, Weight, and E flag are otherwise the same 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|PT |DT |Empty|E|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|               Prioritized Element ID (i)                  ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|               Element Dependency ID (i)                   ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Weight (8)  |
+-+-+-+-+-+-+-+-+
```
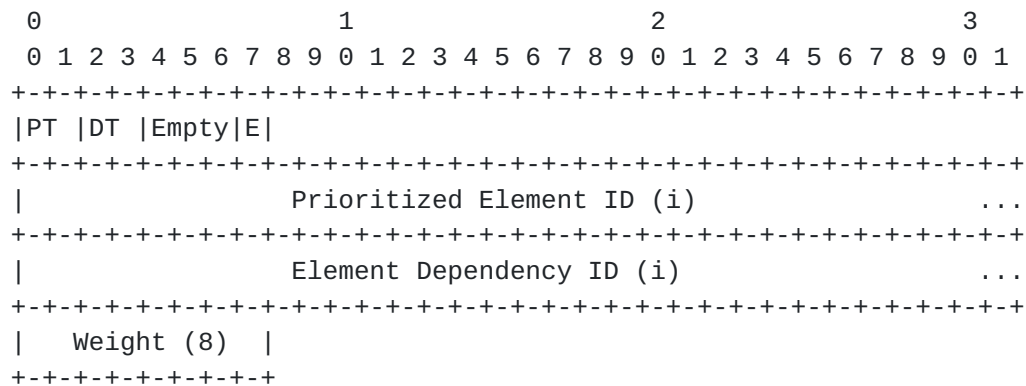
Figure 7: PRIORITY frame payload

The PRIORITY frame payload has the following fields:

Prioritized Type: A two-bit field indicating the type of element
   being prioritized.

Dependency Type: A two-bit field indicating the type of element
   being depended on.

Empty: A three-bit field which MUST be zero when sent and MUST be
   ignored on receipt.

Exclusive: A flag which indicates that the stream dependency is
   exclusive (see [RFC7540], Section 5.3).

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, or a
      Placeholder ID of a placeholder. For details of dependencies, see
      Section 3.2 and [RFC7540], Section 5.3.

   Weight: An unsigned 8-bit integer representing a priority weight for
      the stream (see [RFC7540], Section 5.3). Add one to the value to
      obtain a weight between 1 and 256.

   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 4.2.7), or a placeholder
   using a Placeholder ID (see Section 3.2.1).

   The values for the Prioritized Element Type and Element Dependency
   Type 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 | Ignored            |
         +-----------+------------------+--------------------+
```

   Note that the root of the tree cannot be referenced using a Stream ID
   of 0, as in [RFC7540]; QUIC stream 0 carries a valid HTTP request.
   The root of the tree cannot be reprioritized. A PRIORITY frame that
   prioritizes the root of the tree MUST be treated as a connection
   error of type HTTP_MALFORMED_FRAME.

   When a PRIORITY frame claims to reference a request, the associated
   ID MUST identify a client-initiated bidirectional stream. A server
   MUST treat receipt of PRIORITY frame with a Stream ID of any other
   type as a connection error of type HTTP_MALFORMED_FRAME.

A PRIORITY frame that references a non-existent Push ID or a
Placeholder ID greater than the server's limit MUST be treated as a
HTTP_MALFORMED_FRAME error.

A PRIORITY frame MUST contain only the identified fields. A PRIORITY
frame that contains more or fewer fields, or a PRIORITY frame that
includes a truncated integer encoding MUST be treated as a connection
error of type HTTP_MALFORMED_FRAME.

4.2.5. CANCEL_PUSH

The CANCEL_PUSH frame (type=0x3) is used to request cancellation of
server push prior to the push stream being created. The CANCEL_PUSH
frame identifies a server push request by Push ID (see Section 4.2.7)
using 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 stream. If the push stream has been opened by the
server, the server SHOULD send a QUIC RST_STREAM frame on those
streams and cease transmission of the response.

A server can send this frame to indicate that it won't be sending a
response 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 RST_STREAM frame SHOULD be used instead to
cancel transmission of the server push response.

A CANCEL_PUSH frame is sent on the control stream. Sending a
CANCEL_PUSH frame on a stream other than the control stream MUST be
treated as a stream 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)                         ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```
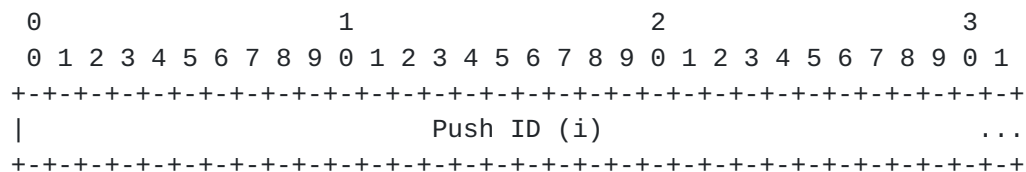
                   Figure 8: CANCEL_PUSH frame payload

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 4.2.7).

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.

An endpoint MUST treat a CANCEL_PUSH frame which does not contain
exactly one properly-formatted variable-length integer as a
connection error of type HTTP_MALFORMED_FRAME.

4.2.6. SETTINGS

The SETTINGS frame (type=0x4) conveys configuration parameters that
affect how endpoints communicate, such as preferences and constraints
on peer behavior, and is different from [RFC7540]. Individually, a
SETTINGS parameter can also be referred to as a "setting".

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 - a peer
uses SETTINGS to advertise a set of supported values. The recipient
can then choose which entries from this list are also acceptable and
proceed with the value it has chosen. (This choice could be
announced in a field of an extension frame, or in its own value in
SETTINGS.)

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. A receiver MAY treat the
presence of the same parameter more than once as a connection error
of type HTTP_MALFORMED_FRAME.

The payload of a SETTINGS frame consists of zero or more parameters,
each consisting of an unsigned 16-bit setting identifier and a value
which uses the QUIC variable-length integer encoding.

```
 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 (16)      |          Value (i)        ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```
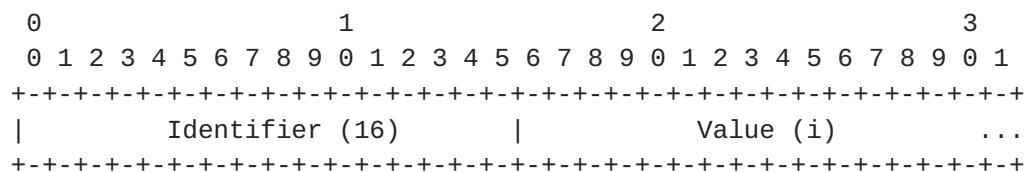
                      Figure 9: SETTINGS value format

Each value MUST be compared against the remaining length of the
SETTINGS frame. Any value which purports to cross the end of the
frame MUST cause the SETTINGS frame to be considered malformed and
trigger a connection error of type HTTP_MALFORMED_FRAME.

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

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

The SETTINGS frame affects connection state. A badly formed or
incomplete SETTINGS frame MUST be treated as a connection error
(Section 6) of type HTTP_MALFORMED_FRAME.

4.2.6.1. Defined SETTINGS Parameters

The following settings are defined in HTTP/QUIC:

SETTINGS_NUM_PLACEHOLDERS (0x3): This value SHOULD be non-zero. The
   default value is 16.

SETTINGS_MAX_HEADER_LIST_SIZE (0x6): The default value is unlimited.

Settings values of the format "0x?a?a" are reserved to exercise the
requirement that unknown parameters 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 MAY be defined by extensions to HTTP/QUIC.

4.2.6.2. Initial SETTINGS Values

When a 0-RTT QUIC connection is being used, the client's initial
requests will be sent before the arrival of the server's SETTINGS
frame. Clients MUST store the settings the server provided in the
session being resumed and MUST comply with stored settings until the
server's current settings are received. Remembered settings apply to
the new connection until the server's SETTINGS frame is received.

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/QUIC
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.

When a 1-RTT QUIC connection is being used, the client MUST NOT send
requests prior to receiving and processing the server's SETTINGS
frame.

4.2.7. PUSH_PROMISE

The PUSH_PROMISE frame (type=0x05) is used to carry a request header
set from server to client, 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 (*)                      ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```
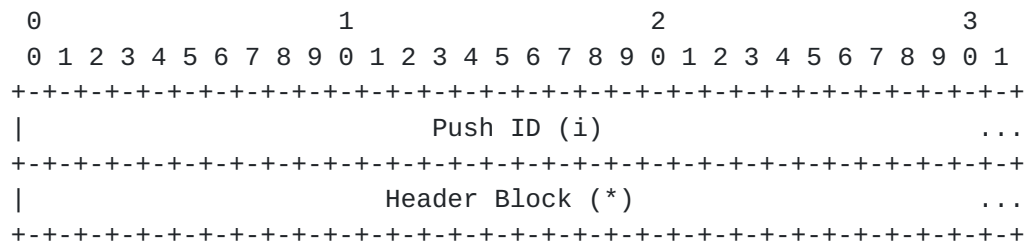
Figure 10: PUSH_PROMISE frame payload

The payload consists of:

Push ID: A variable-length integer that identifies the server push
   request. A push ID is used in push stream header (Section 3.3.3),
   CANCEL_PUSH frames (Section 4.2.5), and PRIORITY frames
   (Section 4.2.4).

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 4.2.9). A client MUST treat
receipt of a PUSH_PROMISE that contains a larger Push ID than the
client has advertised as a connection error of type
HTTP_MALFORMED_FRAME.

A server MAY use the same Push ID in multiple PUSH_PROMISE frames.
This allows the server to use the same server push in response to
multiple concurrent requests. Referencing the same server push
ensures that a PUSH_PROMISE can be made in relation to every response
in which server push might be needed without duplicating pushes.

A server that uses the same Push ID in multiple PUSH_PROMISE frames
MUST include the same header fields each time. The octets of the

header block MAY be different due to differing encoding, but the
header fields and their values MUST be identical. Note that ordering
of header fields is significant. A client MUST treat receipt of a
PUSH_PROMISE with conflicting header field values for the same Push
ID as a connection error of type HTTP_MALFORMED_FRAME.

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

4.2.8. 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)                        ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```
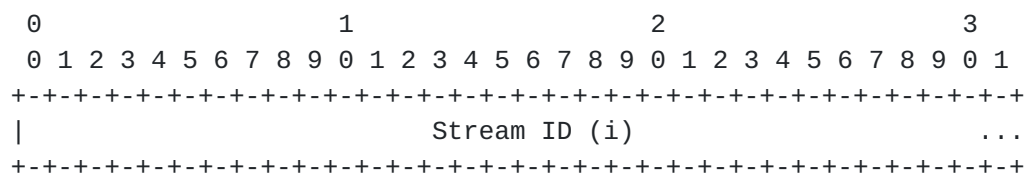
Figure 11: GOAWAY frame payload

The GOAWAY frame 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 as a connection error (Section 6) of type
HTTP_UNEXPECTED_GOAWAY.

The GOAWAY frame applies to the connection, not a specific stream.
An endpoint MUST treat a GOAWAY frame on a stream other than the
control stream as a connection error (Section 6) of type
HTTP_WRONG_STREAM.

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

4.2.9. 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_STREAM_ID frame.

The MAX_PUSH_ID frame is always sent on a 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_MALFORMED_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 a MAX_PUSH_ID frame 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)                         ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```
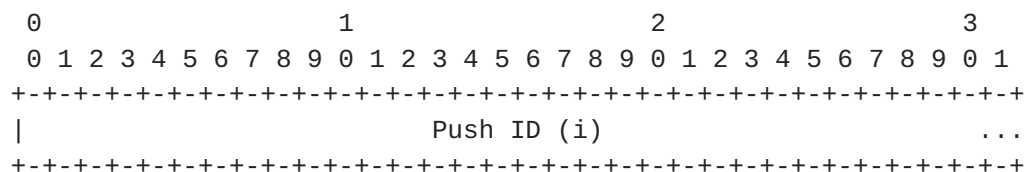
                    Figure 12: MAX_PUSH_ID frame payload

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 4.2.7). 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_MALFORMED_FRAME.

A server MUST treat a MAX_PUSH_ID frame payload that does not contain
a single variable-length integer as a connection error of type
HTTP_MALFORMED_FRAME.

5. Connection Closure

Once established, an HTTP/QUIC 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/QUIC 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 use QUIC PING frames to keep connections open while there are responses outstanding for requests or server pushes. 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 use PING to maintain connections in anticipation of need rather than incur the latency cost of connection establishment to servers. Servers SHOULD NOT use PING frames to keep a connection 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 4.2.8). 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 not accepted. This enables client and server to agree on which requests were accepted prior to the connection shutdown. This identifier MAY be lower than the stream limit identified by a QUIC MAX_STREAM_ID frame, and MAY be zero if no requests were processed. Servers SHOULD NOT increase the QUIC MAX_STREAM_ID limit after sending a GOAWAY frame.

Once sent, the server MUST cancel requests sent on streams with an identifier higher than the indicated last Stream ID. Clients MUST NOT send new requests on the connection after receiving GOAWAY, although requests might already be in transit. A new connection can be established for new requests.

If the client has sent requests on streams with a higher Stream ID than indicated in the GOAWAY frame, those requests are considered

cancelled (Section 3.1.3). Clients SHOULD reset any streams above
this ID with the error code HTTP_REQUEST_CANCELLED. Servers MAY also
cancel requests on streams below the indicated ID if these requests
were not processed.

Requests on Stream IDs less than the Stream ID in the GOAWAY frame
might have been processed; their status cannot be known until they
are completed successfully, reset individually, or the connection
terminates.

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 stream 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 current value of QUIC's MAX_STREAM_ID and SHOULD NOT
increase the MAX_STREAM_ID 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.

5.3. Immediate Application Closure

An HTTP/QUIC implementation can immediately close the QUIC connection
at any time. This results in sending a QUIC APPLICATION_CLOSE frame
to the peer; the error code in this frame indicates to the peer why
the connection is being closed. See Section 6 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 APPLICATION_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. 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].

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

6.1. HTTP/QUIC Error Codes

The following error codes are defined for use in QUIC RST_STREAM,
STOP_SENDING, and APPLICATION_CLOSE frames when using HTTP/QUIC.

STOPPING (0x00): This value is reserved by the transport to be used
   in response to QUIC STOP_SENDING frames.

HTTP_NO_ERROR (0x01): No error. This is used when the connection or
   stream needs to be closed, but there is no error to signal.

HTTP_PUSH_REFUSED (0x02): The server has attempted to push content
   which the client will not accept on this connection.

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

HTTP_PUSH_ALREADY_IN_CACHE (0x04): The server has attempted to push
   content which the client has cached.

HTTP_REQUEST_CANCELLED (0x05): The client no longer needs the
   requested data.

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/QUIC. 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_PUSH_LIMIT_EXCEEDED (0x0B): A Push ID greater than the current
    maximum Push ID was referenced.

HTTP_DUPLICATE_PUSH (0x0C): A Push ID was referenced in two
    different stream headers.

HTTP_UNKNOWN_STREAM_TYPE (0x0D): A unidirectional stream header
    contained an unknown stream type.

HTTP_WRONG_STREAM_COUNT (0x0E): A unidirectional stream type was
    used more times than is permitted by that type.

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

HTTP_WRONG_STREAM_DIRECTION (0x0010): A unidirectional stream type
    was used by a peer which is not permitted to do so.

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_GENERAL_PROTOCOL_ERROR (0x00FF): 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.

HTTP_MALFORMED_FRAME (0x01XX): An error in a specific frame type.
    The frame type is included as the last octet of the error code.
    For example, an error in a MAX_PUSH_ID frame would be indicated
    with the code (0x10D).

7. Extensions to HTTP/QUIC

   HTTP/QUIC 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/QUIC 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 4.2), new
   settings (Section 4.2.6.1), new error codes (Section 6), or new
   unidirectional stream types (Section 3.3). Registries are
   established for managing these extension points: frame types
   (Section 10.3), settings (Section 10.4), error codes (Section 10.5),
   and stream types (Section 10.6).

   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 4.2.6.1) 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.

8. Considerations for Transitioning from HTTP/2

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

   HTTP/QUIC begins from the premise that HTTP/2 code reuse is a useful
   feature, but not a hard requirement. HTTP/QUIC departs from HTTP/2

primarily where necessary to accommodate the differences in behavior
between QUIC and TCP (lack of ordering, support for streams). We
intend to avoid gratuitous changes which make it difficult or
impossible to build extensions with the same semantics applicable to
both protocols at once.

These departures are noted in this section.

8.1. Streams

HTTP/QUIC 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.

8.2. HTTP Frame Types

Many framing concepts from HTTP/2 can be elided away 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 [RFC7540]. 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/
QUIC. 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/QUIC 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/QUIC will break them.

For example, 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. HTTP/2 specifies

   priority assignments in PRIORITY frames and (optionally) in HEADERS
   frames. To achieve in-order delivery of priority changes in HTTP/
   QUIC, PRIORITY frames are sent on the control stream and the PRIORITY
   section is removed from the HEADERS frame.

   Likewise, 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. As a result, HTTP/QUIC uses a modified version of HPACK,
   described in [QPACK].

   Frame type definitions in HTTP/QUIC often use the QUIC variable-
   length integer encoding. In particular, Stream IDs use this
   encoding, which allow for a larger range of possible values than the
   encoding used in HTTP/2. Some frames in HTTP/QUIC 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/QUIC 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/QUIC. HTTP/QUIC extensions will not assume
   ordering, but would not be harmed by ordering, and would be portable
   to HTTP/2 in the same manner.

   Below is a listing of how each HTTP/2 frame type is mapped:

   DATA (0x0): Padding is not defined in HTTP/QUIC frames. See
      Section 4.2.2.

   HEADERS (0x1): As described above, the PRIORITY region of HEADERS is
      not supported. A separate PRIORITY frame MUST be used. Padding
      is not defined in HTTP/QUIC frames. See Section 4.2.3.

   PRIORITY (0x2): As described above, the PRIORITY frame is sent on
      the control stream and can reference either a Stream ID or a Push
      ID. See Section 4.2.4.

   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 4.2.5).

SETTINGS (0x4): SETTINGS frames are sent only at the beginning of
   the connection. See Section 4.2.6 and Section 8.3.

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 4.2.7.

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 4.2.8.

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, and
   HEADERS frames can be used in series.

Frame types defined by extensions to HTTP/2 need to be separately
registered for HTTP/QUIC if still applicable. The IDs of frames
defined in [RFC7540] have been reserved for simplicity. See
Section 10.3.

8.3. 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/QUIC. The
HTTP-level options that are retained in HTTP/QUIC 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 Section 4.2.6.1.

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/
   QUIC. Specifying it in the SETTINGS frame is an error.

SETTINGS_MAX_HEADER_LIST_SIZE: See Section 4.2.6.1.

In HTTP/QUIC, 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/QUIC. The
IDs of settings defined in [RFC7540] have been reserved for
simplicity. See Section 10.4.

8.4. HTTP/2 Error Codes

QUIC has the same concepts of "stream" and "connection" errors that
HTTP/2 provides. However, because the error code space is shared
between multiple components, there is no direct portability of HTTP/2
error codes.

The HTTP/2 error codes defined in Section 7 of [RFC7540] map to the
HTTP/QUIC error codes as follows:

NO_ERROR (0x0): HTTP_NO_ERROR in Section 6.1.

PROTOCOL_ERROR (0x1): No single mapping. See new
   HTTP_MALFORMED_FRAME error codes defined in Section 6.1.

INTERNAL_ERROR (0x2): HTTP_INTERNAL_ERROR in Section 6.1.

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): No single mapping. See new error codes
   defined in Section 6.1.

REFUSED_STREAM (0x7): Not applicable, since QUIC handles stream
   management. Would provoke a QUIC_TOO_MANY_OPEN_STREAMS from the
   QUIC layer.

CANCEL (0x8): HTTP_REQUEST_CANCELLED in Section 6.1.

COMPRESSION_ERROR (0x9): HTTP_QPACK_DECOMPRESSION_FAILED in [QPACK].

CONNECT_ERROR (0xa): HTTP_CONNECT_ERROR in Section 6.1.

ENHANCE_YOUR_CALM (0xb): HTTP_EXCESSIVE_LOAD in Section 6.1.

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 6.1.

Error codes need to be defined for HTTP/2 and HTTP/QUIC separately.
See Section 10.5.

9. Security Considerations

The security considerations of HTTP/QUIC should be comparable to
those of HTTP/2 with TLS. Note that where HTTP/2 employs PADDING
frames to make a connection more resistant to traffic analysis, HTTP/
QUIC can rely on QUIC's own PADDING frames or employ the reserved
frame and stream types discussed in Section 4.2.1 and Section 3.3.1.

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

The modified SETTINGS format contains nested length elements, which
could pose a security risk to an incautious implementer. A SETTINGS
frame parser MUST ensure that the length of the frame exactly matches
the length of the settings it contains.

10. IANA Considerations

10.1. Registration of HTTP/QUIC Identification String

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

The "hq" string identifies HTTP/QUIC:

   Protocol: HTTP/QUIC

   Identification Sequence: 0x68 0x71 ("hq")

   Specification: This document

10.2. 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 2.2.1

10.3. Frame Types

   This document establishes a registry for HTTP/QUIC frame type codes.
   The "HTTP/QUIC Frame Type" registry manages an 8-bit space. The
   "HTTP/QUIC Frame Type" registry operates under either of the "IETF
   Review" or "IESG Approval" policies [RFC8126] for values from 0x00 up
   to and including 0xef, with values from 0xf0 up to and including 0xff
   being reserved for Experimental Use.

   While this registry is separate from the "HTTP/2 Frame Type" registry
   defined in [RFC7540], 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 entries in this registry require the following information:

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

   Code: The 8-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 4.2.2 |
        |              |      |               |
        | HEADERS      | 0x1  | Section 4.2.3 |
        |              |      |               |
        | PRIORITY     | 0x2  | Section 4.2.4 |
        |              |      |               |
        | CANCEL_PUSH  | 0x3  | Section 4.2.5 |
        |              |      |               |
        | SETTINGS     | 0x4  | Section 4.2.6 |
        |              |      |               |
        | PUSH_PROMISE | 0x5  | Section 4.2.7 |
        |              |      |               |
        | Reserved     | 0x6  | N/A           |
        |              |      |               |
        | GOAWAY       | 0x7  | Section 4.2.8 |
        |              |      |               |
        | Reserved     | 0x8  | N/A           |
        |              |      |               |
        | Reserved     | 0x9  | N/A           |
        |              |      |               |
        | MAX_PUSH_ID  | 0xD  | Section 4.2.9 |
        +--------------+------+---------------+
```

Additionally, each code of the format "0xb + (0x1f * N)" for values of N in the range (0..7) (that is, "0xb", "0x2a", "0x49", "0x68", "0x87", "0xa6", "0xc5", and "0xe4"), the following values should be registered:

Frame Type: Reserved - GREASE

Specification: Section 4.2.1

10.4. Settings Parameters

This document establishes a registry for HTTP/QUIC settings. The "HTTP/QUIC Settings" registry manages a 16-bit space. The "HTTP/QUIC Settings" registry operates under the "Expert Review" policy [RFC8126] for values in the range from 0x0000 to 0xefff, with values between and 0xf000 and 0xffff being reserved for Experimental Use. The designated experts are the same as those for the "HTTP/2 Settings" registry defined in [RFC7540].

While this registry is separate from the "HTTP/2 Settings" registry defined in [RFC7540], 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 16-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             |
|                      |      |                 |
| NUM_PLACEHOLDERS     | 0x3  | Section 4.2.6.1 |
|                      |      |                 |
| Reserved             | 0x4  | N/A             |
|                      |      |                 |
| Reserved             | 0x5  | N/A             |
|                      |      |                 |
| MAX_HEADER_LIST_SIZE | 0x6  | Section 4.2.6.1 |
+----------------------+------+-----------------+
```

Additionally, each code of the format "0x?a?a" where each "?" is any
four bits (that is, "0x0a0a", "0x0a1a", etc. through "0xfafa"), the
following values should be registered:

Name: Reserved - GREASE

Specification: Section 4.2.6.1

10.5. Error Codes

This document establishes a registry for HTTP/QUIC error codes. The
"HTTP/QUIC Error Code" registry manages a 16-bit space. The "HTTP/
QUIC 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 16-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
STOPPING	0x0000	Reserved by QUIC	[QUIC-TRANSPORT]
HTTP_NO_ERROR	0x0001	No error	Section 6.1
HTTP_PUSH_REFUSED	0x0002	Client refused pushed content	Section 6.1
HTTP_INTERNAL_ERROR	0x0003	Internal error	Section 6.1
HTTP_PUSH_ALREADY_IN_CACHE	0x0004	Pushed content already cached	Section 6.1
HTTP_REQUEST_CANCELLED	0x0005	Data no longer needed	Section 6.1
HTTP_INCOMPLETE_REQUEST	0x0006	Stream terminated early	Section 6.1
HTTP_CONNECT_ERROR	0x0007	TCP reset or error on CONNECT request	Section 6.1

HTTP_EXCESSIVE_LOAD	0x0008	Peer generating excessive load	Section 6.1
HTTP_VERSION_FALLBACK	0x0009	Retry over HTTP/1.1	Section 6.1
HTTP_WRONG_STREAM	0x000A	A frame was sent on the wrong stream	Section 6.1
HTTP_PUSH_LIMIT_EXCEEDED	0x000B	Maximum Push ID exceeded	Section 6.1
HTTP_DUPLICATE_PUSH	0x000C	Push ID was fulfilled multiple times	Section 6.1
HTTP_UNKNOWN_STREAM_TYPE	0x000D	Unknown unidirectional stream type	Section 6.1
HTTP_WRONG_STREAM_COUNT	0x000E	Too many unidirectional streams	Section 6.1
HTTP_CLOSED_CRITICAL_STREAM	0x000F	Critical stream was closed	Section 6.1
HTTP_WRONG_STREAM_DIRECTION	0x0010	Unidirectional stream in wrong direction	Section 6.1
HTTP_EARLY_RESPONSE	0x0011	Remainder of request not needed	Section 6.1
HTTP_MISSING_SETTINGS	0x0012	No SETTINGS frame received	Section 6.1
HTTP_MALFORMED_FRAME	0x01XX	Error in frame	Section 6.1

```
    |                         |       | formatting or |                 |
    |                         |       | use           |                 |
    +-------------------------+-------+---------------+-----------------+
```

10.6. Stream Types

   This document establishes a registry for HTTP/QUIC unidirectional
   stream types. The "HTTP/QUIC Stream Type" registry manages an 8-bit
   space. The "HTTP/QUIC Stream Type" registry operates under either of
   the "IETF Review" or "IESG Approval" policies [RFC8126] for values
   from 0x00 up to and including 0xef, with values from 0xf0 up to and
   including 0xff being reserved for Experimental Use.

   New entries in this registry require the following information:

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

   Code: The 8-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 | 0x43 | Section 3.3.2 | Both   |
         |                |      |               |        |
         | Push Stream    | 0x50 | Section 3.3.3 | Server |
         +----------------+------+---------------+--------+
```

   Additionally, for each code of the format "0x1f * N" for values of N
   in the range (0..8) (that is, "0x00", "0x1f", "0x3e", "0x5d", "0x7c",
   "0x9b", "0xba", "0xd9", "0xf8"), the following values should be
   registered:

   Stream Type: Reserved - GREASE

   Specification: Section 3.3.1

   Sender: Both

11. References

11.1. 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>.

[QPACK]   Krasic, C., Bishop, M., and A. Frindell, Ed., "QPACK:
           Header Compression for HTTP over QUIC", draft-ietf-quic-
           qpack-03 (work in progress), October 2018.

[QUIC-TRANSPORT]
           Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based
           Multiplexed and Secure Transport", draft-ietf-quic-
           transport-14 (work in progress), October 2018.

[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>.

   [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>.

11.2. Informative References

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

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

11.3. 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. Change Log

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

A.1. 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)

A.2. **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)

A.3. **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/QUIC 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)

A.4. **Since draft-ietf-quic-http-11**

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

A.5. **Since draft-ietf-quic-http-10**

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

A.6. 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)

A.7. Since draft-ietf-quic-http-08

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

A.8. 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)

A.9. Since draft-ietf-quic-http-06

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

A.10. 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)

A.11. 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)

A.12. Since draft-ietf-quic-http-03

   None.

A.13. Since draft-ietf-quic-http-02

o Track changes in transport draft

A.14. 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)

A.15. 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)

A.16. 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