                 Hypertext Transfer Protocol Version 2 (HTTP/2)
                        draft-ietf-httpbis-http2bis-02

Abstract

   This specification describes an optimized expression of the semantics
   of the Hypertext Transfer Protocol (HTTP), referred to as HTTP
   version 2 (HTTP/2). HTTP/2 enables a more efficient use of network
   resources and a reduced perception of latency by introducing header
   field compression and allowing multiple concurrent exchanges on the
   same connection.

   This specification is an alternative to, but does not obsolete, the
   HTTP/1.1 message syntax. HTTP's existing semantics remain unchanged.
   This document obsoletes RFC 7540 and RFC 8740.

Discussion Venues

   This note is to be removed before publishing as an RFC.

   Discussion of this document takes place on the HTTPBIS Working Group
   mailing list (ietf-http-wg@w3.org), which is archived at
   https://lists.w3.org/Archives/Public/ietf-http-wg/.

   Source for this draft and an issue tracker can be found at
   https://github.com/httpwg/http2-spec.

Status of This Memo

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

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

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

This Internet-Draft will expire on 4 December 2021.

Copyright Notice

Table of Contents

1. Introduction

   The Hypertext Transfer Protocol (HTTP) is a wildly successful
   protocol. However, the way HTTP/1.1 uses the underlying transport
   ([HTTP11]) has several characteristics that have a negative overall
   effect on application performance today.

   In particular, HTTP/1.0 allowed only one request to be outstanding at
   a time on a given TCP connection. HTTP/1.1 added request pipelining,
   but this only partially addressed request concurrency and still
   suffers from head-of-line blocking. Therefore, HTTP/1.0 and HTTP/1.1
   clients that need to make many requests use multiple connections to a
   server in order to achieve concurrency and thereby reduce latency.

Furthermore, HTTP header fields are often repetitive and verbose, causing unnecessary network traffic as well as causing the initial TCP [TCP] congestion window to quickly fill. This can result in excessive latency when multiple requests are made on a new TCP connection.

HTTP/2 addresses these issues by defining an optimized mapping of HTTP's semantics to an underlying connection. Specifically, it allows interleaving of request and response messages on the same connection and uses an efficient coding for HTTP header fields. It also allows prioritization of requests, letting more important requests complete more quickly, further improving performance.

The resulting protocol is more friendly to the network because fewer TCP connections can be used in comparison to HTTP/1.x. This means less competition with other flows and longer-lived connections, which in turn lead to better utilization of available network capacity.

Finally, HTTP/2 also enables more efficient processing of messages through use of binary message framing.

This document obsoletes RFC 7540 [RFC7540] and RFC 8740 [RFC8740].

2. HTTP/2 Protocol Overview

HTTP/2 provides an optimized transport for HTTP semantics. HTTP/2 supports all of the core features of HTTP but aims to be more efficient than HTTP/1.1.

The basic protocol unit in HTTP/2 is a frame (Section 4.1). Each frame type serves a different purpose. For example, HEADERS and DATA frames form the basis of HTTP requests and responses (Section 8.1); other frame types like SETTINGS, WINDOW_UPDATE, and PUSH_PROMISE are used in support of other HTTP/2 features.

Multiplexing of requests is achieved by having each HTTP request/ response exchange associated with its own stream (Section 5). Streams are largely independent of each other, so a blocked or stalled request or response does not prevent progress on other streams.

Flow control and prioritization ensure that it is possible to efficiently use multiplexed streams. Flow control (Section 5.2) helps to ensure that only data that can be used by a receiver is transmitted. Prioritization (Section 5.3) ensures that limited resources can be directed to the most important streams first.

Because HTTP header fields used in a connection can contain large amounts of redundant data, frames that contain them are compressed (Section 4.3). This has especially advantageous impact upon request sizes in the common case, allowing many requests to be compressed into one packet.

Finally, HTTP/2 adds a new, optional interaction mode whereby a server can push responses to a client (Section 8.2). This is intended to allow a server to speculatively send data to a client that the server anticipates the client will need, trading off some network usage against a potential latency gain. The server does this by synthesizing a request, which it sends as a PUSH_PROMISE frame. The server is then able to send a response to the synthetic request on a separate stream.

2.1. Document Organization

The HTTP/2 specification is split into four parts:

* Starting HTTP/2 (Section 3) covers how an HTTP/2 connection is initiated.

* The frame (Section 4) and stream (Section 5) layers describe the way HTTP/2 frames are structured and formed into multiplexed streams.

* Frame (Section 6) and error (Section 7) definitions include details of the frame and error types used in HTTP/2.

* HTTP mappings (Section 8) and additional requirements (Section 9) describe how HTTP semantics are expressed using frames and streams.

While some of the frame and stream layer concepts are isolated from

HTTP, this specification does not define a completely generic frame
layer. The frame and stream layers are tailored to the needs of the
HTTP protocol and server push.

2.2. Conventions and Terminology

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

All numeric values are in network byte order. Values are unsigned
unless otherwise indicated. Literal values are provided in decimal
or hexadecimal as appropriate. Hexadecimal literals are prefixed
with "0x" to distinguish them from decimal literals.

The following terms are used:

client: The endpoint that initiates an HTTP/2 connection. Clients
   send HTTP requests and receive HTTP responses.

connection: A transport-layer connection between two endpoints.

connection error: An error that affects the entire HTTP/2
   connection.

endpoint: Either the client or server of the connection.

frame: The smallest unit of communication within an HTTP/2
   connection, consisting of a header and a variable-length sequence
   of octets structured according to the frame type.

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/2 connection. Servers

receive HTTP requests and send HTTP responses.

   stream: A bidirectional flow of frames within the HTTP/2 connection.

   stream error: An error on the individual HTTP/2 stream.

   Finally, the terms "gateway", "intermediary", "proxy", and "tunnel"
   are defined in Section 3.7 of [HTTP]. Intermediaries act as both
   client and server at different times.

   The term "content" as it applies to message bodies is defined in
   Section 6.4 of [HTTP].

3. Starting HTTP/2

   An HTTP/2 connection is an application-layer protocol running on top
   of a TCP connection ([TCP]). The client is the TCP connection
   initiator.

   HTTP/2 uses the "http" and "https" URI schemes defined in Section 4.2
   of [HTTP]. HTTP/2 shares the same default port numbers: 80 for
   "http" URIs and 443 for "https" URIs. As a result, implementations
   processing requests for target resource URIs like
   "http://example.org/foo" or "https://example.com/bar" are required to
   first discover whether the upstream server (the immediate peer to
   which the client wishes to establish a connection) supports HTTP/2.

   The means by which support for HTTP/2 is determined is different for
   "http" and "https" URIs. Discovery for "https" URIs is described in
   Section 3.2. HTTP/2 support for "http" URIs can only be discovered
   by out-of-band means, and requires prior knowledge of the support as
   described in Section 3.3.

3.1. HTTP/2 Version Identification

   The protocol defined in this document has two identifiers.

   * The string "h2" identifies the protocol where HTTP/2 uses

Transport Layer Security (TLS) [TLS13]. This identifier is used
        in the TLS application-layer protocol negotiation (ALPN) extension
        [TLS-ALPN] field and in any place where HTTP/2 over TLS is
        identified.

        The "h2" string is serialized into an ALPN protocol identifier as
        the two-octet sequence: 0x68, 0x32.

   *   The string "h2c" identifies the protocol where HTTP/2 is run over
        cleartext TCP. This identifier is used in any place where HTTP/2
        over TCP is identified.

        The "h2c" string is reserved from the ALPN identifier space but
        describes a protocol that does not use TLS.

        The "h2c" string was previously used as a token for use in the
        HTTP Upgrade mechanism's Upgrade header field (Section 7.8 of
        [HTTP]). This usage was never widely deployed, and is no longer
        specified in this document.

   Negotiating "h2" or "h2c" implies the use of the transport, security,
   framing, and message semantics described in this document.

3.2. Starting HTTP/2 for "https" URIs

   A client that makes a request to an "https" URI uses TLS [TLS13] with
   the application-layer protocol negotiation (ALPN) extension
   [TLS-ALPN].

   HTTP/2 over TLS uses the "h2" protocol identifier. The "h2c"
   protocol identifier MUST NOT be sent by a client or selected by a
   server; the "h2c" protocol identifier describes a protocol that does
   not use TLS.

   Once TLS negotiation is complete, both the client and the server MUST
   send a connection preface (Section 3.4).

3.3. Starting HTTP/2 with Prior Knowledge

   A client can learn that a particular server supports HTTP/2 by other
   means. For example, [ALT-SVC] describes a mechanism for advertising
   this capability.

A client MUST send the connection preface (Section 3.4) and then MAY
immediately send HTTP/2 frames to such a server; servers can identify
these connections by the presence of the connection preface. This
only affects the establishment of HTTP/2 connections over cleartext
TCP; implementations that support HTTP/2 over TLS MUST use protocol
negotiation in TLS [TLS-ALPN].

Likewise, the server MUST send a connection preface (Section 3.4).

Without additional information, prior support for HTTP/2 is not a
strong signal that a given server will support HTTP/2 for future
connections. For example, it is possible for server configurations
to change, for configurations to differ between instances in
clustered servers, or for network conditions to change.

3.4. HTTP/2 Connection Preface

In HTTP/2, each endpoint is required to send a connection preface as
a final confirmation of the protocol in use and to establish the
initial settings for the HTTP/2 connection. The client and server
each send a different connection preface.

The client connection preface starts with a sequence of 24 octets,
which in hex notation is:

   0x505249202a20485454502f322e300d0a0d0a534d0d0a0d0a

That is, the connection preface starts with the string "PRI *
HTTP/2.0\r\n\r\nSM\r\n\r\n". This sequence MUST be followed by a
SETTINGS frame (Section 6.5), which MAY be empty. The client sends
the client connection preface as the first application data octets of
a connection.

   | Note: The client connection preface is selected so that a large
   | proportion of HTTP/1.1 or HTTP/1.0 servers and intermediaries
   | do not attempt to process further frames. Note that this does
   | not address the concerns raised in [TALKING].

The server connection preface consists of a potentially empty
SETTINGS frame (Section 6.5) that MUST be the first frame the server
sends in the HTTP/2 connection.

The SETTINGS frames received from a peer as part of the connection
preface MUST be acknowledged (see Section 6.5.3) after sending the
connection preface.

   To avoid unnecessary latency, clients are permitted to send
   additional frames to the server immediately after sending the client
   connection preface, without waiting to receive the server connection
   preface. It is important to note, however, that the server
   connection preface SETTINGS frame might include settings that
   necessarily alter how a client is expected to communicate with the
   server. Upon receiving the SETTINGS frame, the client is expected to
   honor any settings established. In some configurations, it is
   possible for the server to transmit SETTINGS before the client sends
   additional frames, providing an opportunity to avoid this issue.

   Clients and servers MUST treat an invalid connection preface as a
   connection error (Section 5.4.1) of type PROTOCOL_ERROR. A GOAWAY
   frame (Section 6.8) MAY be omitted in this case, since an invalid
   preface indicates that the peer is not using HTTP/2.

4. HTTP Frames

   Once the HTTP/2 connection is established, endpoints can begin
   exchanging frames.

4.1. Frame Format

   All frames begin with a fixed 9-octet header followed by a variable-
   length frame payload.

```
    +-----------------------------------------------+
    |                 Length (24)                   |
    +---------------+---------------+---------------+
    |   Type (8)    |   Flags (8)   |
    +-+-------------+---------------+-------------------------------+
    |R|                 Stream Identifier (31)                     |
    +=+=============================================================+
    |                   Frame Payload (0...)                     ...
    +---------------------------------------------------------------+
```

                        Figure 1: Frame Layout

   The fields of the frame header are defined as:

Length: The length of the frame payload expressed as an unsigned
   24-bit integer. Values greater than 2^14 (16,384) MUST NOT be
   sent unless the receiver has set a larger value for
   SETTINGS_MAX_FRAME_SIZE.

   The 9 octets of the frame header are not included in this value.

Type: The 8-bit type of the frame. The frame type determines the

   format and semantics of the frame. Implementations MUST ignore
   and discard any frame that has a type that is unknown.

Flags: An 8-bit field reserved for boolean flags specific to the
   frame type.

   Flags are assigned semantics specific to the indicated frame type.
   Flags that have no defined semantics for a particular frame type
   MUST be ignored and MUST be left unset (0x0) when sending.

R: A reserved 1-bit field. The semantics of this bit are undefined,
   and the bit MUST remain unset (0x0) when sending and MUST be
   ignored when receiving.

Stream Identifier: A stream identifier (see Section 5.1.1) expressed
   as an unsigned 31-bit integer. The value 0x0 is reserved for
   frames that are associated with the connection as a whole as
   opposed to an individual stream.

   The structure and content of the frame payload is dependent entirely
   on the frame type.

4.2. Frame Size

   The size of a frame payload is limited by the maximum size that a
   receiver advertises in the SETTINGS_MAX_FRAME_SIZE setting. This
   setting can have any value between 2^14 (16,384) and 2^24-1
   (16,777,215) octets, inclusive.

   All implementations MUST be capable of receiving and minimally
   processing frames up to 2^14 octets in length, plus the 9-octet frame
   header (Section 4.1). The size of the frame header is not included

when describing frame sizes.

> | Note: Certain frame types, such as PING (Section 6.7), impose
> | additional limits on the amount of frame payload data allowed.

An endpoint MUST send an error code of FRAME_SIZE_ERROR if a frame
exceeds the size defined in SETTINGS_MAX_FRAME_SIZE, exceeds any
limit defined for the frame type, or is too small to contain
mandatory frame data. A frame size error in a frame that could alter
the state of the entire connection MUST be treated as a connection
error (Section 5.4.1); this includes any frame carrying a field block
(Section 4.3) (that is, HEADERS, PUSH_PROMISE, and CONTINUATION),
SETTINGS, and any frame with a stream identifier of 0.

Endpoints are not obligated to use all available space in a frame.
Responsiveness can be improved by using frames that are smaller than
the permitted maximum size. Sending large frames can result in
delays in sending time-sensitive frames (such as RST_STREAM,
WINDOW_UPDATE, or PRIORITY), which, if blocked by the transmission of
a large frame, could affect performance.

4.3. Field Section Compression and Decompression

Field section compression is the process of compressing a set of
field lines to form a field block. Field section decompression is
the process of decoding a field block into a set of field lines.
Details of HTTP/2 field section compression and decompression is
defined in [COMPRESSION], which, for historical reasons, refers to
these processes as header compression and decompression.

Field blocks carry the compressed bytes of a field section, with
header sections also carrying control data associated with the
message in the form of pseudo-header fields (Section 8.1.2.1) that
use the same format as a field line.

> | Note: Previous versions of this specification used the term
> | "header block" in place of the more generic "field block".

Field blocks carry control data and header sections for requests,

responses, promised requests, and pushed responses (see Section 8.2).
All these messages, except for interim responses and requests
contained in PUSH_PROMISE (Section 6.6) frames can optionally include
a field block that carries a trailer section.

A field section is a collection of zero or more field lines. Each of
the field lines in a field block carry a single value. The
serialized field block is then divided into one or more octet
sequences, called field block fragments, and transmitted within the
frame payload of HEADERS (Section 6.2) or PUSH_PROMISE (Section 6.6),
each of which could be followed by CONTINUATION (Section 6.10)
frames.

The Cookie header field [COOKIE] is treated specially by the HTTP
mapping (see Section 8.1.2.5).

A receiving endpoint reassembles the field block by concatenating its
fragments and then decompresses the block to reconstruct the field
section.

A complete field section consists of either:

   * a single HEADERS or PUSH_PROMISE frame, with the END_HEADERS flag
      set, or

   * a HEADERS or PUSH_PROMISE frame with the END_HEADERS flag cleared
      and one or more CONTINUATION frames, where the last CONTINUATION
      frame has the END_HEADERS flag set.

Field compression is stateful. One compression context and one
decompression context are used for the entire connection. A decoding
error in a field block MUST be treated as a connection error
(Section 5.4.1) of type COMPRESSION_ERROR.

Each field block is processed as a discrete unit. Field blocks MUST
be transmitted as a contiguous sequence of frames, with no
interleaved frames of any other type or from any other stream. The
last frame in a sequence of HEADERS or CONTINUATION frames has the
END_HEADERS flag set. The last frame in a sequence of PUSH_PROMISE
or CONTINUATION frames has the END_HEADERS flag set. This allows a

field block to be logically equivalent to a single frame.

Field block fragments can only be sent as the frame payload of
HEADERS, PUSH_PROMISE, or CONTINUATION frames because these frames
carry data that can modify the compression context maintained by a
receiver. An endpoint receiving HEADERS, PUSH_PROMISE, or
CONTINUATION frames needs to reassemble field blocks and perform
decompression even if the frames are to be discarded. A receiver
MUST terminate the connection with a connection error (Section 5.4.1)
of type COMPRESSION_ERROR if it does not decompress a field block.

5. Streams and Multiplexing

A "stream" is an independent, bidirectional sequence of frames
exchanged between the client and server within an HTTP/2 connection.
Streams have several important characteristics:

* A single HTTP/2 connection can contain multiple concurrently open
   streams, with either endpoint interleaving frames from multiple
   streams.

* Streams can be established and used unilaterally or shared by
   either the client or server.

* Streams can be closed by either endpoint.

* The order in which frames are sent on a stream is significant.
   Recipients process frames in the order they are received. In
   particular, the order of HEADERS and DATA frames is semantically
   significant.

* Streams are identified by an integer. Stream identifiers are
   assigned to streams by the endpoint initiating the stream.

5.1. Stream States

The lifecycle of a stream is shown in Figure 2.

```
                        +--------+
                send PP |        | recv PP
               ,--------|  idle  |--------.
              /         |        |         \
```

```
                 v      +--------+          v
            +---------+  |        |     +---------+
            |         |  | send H /|     |         |
      ,------| reserved |  | recv H  |     | reserved |------.
      |      | (local) |  |        |     | (remote) |      |
      |      +---------+  v        |     +---------+      |
      |          |     +--------+  |         |          |
      |          |     | recv ES |  | send ES  |          |
      |  send H  |   ,------|  open  |------.   | recv H  |
      |          |  /  |        |  |     \  |          |
      |          v v   +--------+  |      v v          |
      |      +---------+  |        |     +---------+      |
      |      |  half   |  |        |     |  half   |      |
      |      | closed  |  | send R /|     | closed  |      |
      |      | (remote) |  | recv R  |     | (local) |      |
      |      +---------+  |        |     +---------+      |
      |          |     |        |         |          |
      |          | send ES /  |         recv ES / |          |
      |          | send R /   v         send R /  |          |
      |          | recv R   +--------+   recv R    |          |
      | send R /  `---------->|        |<-----------'  send R / |
      | recv R            | closed |             recv R  |
      `----------------------->|        |<----------------------'
                          +--------+
```
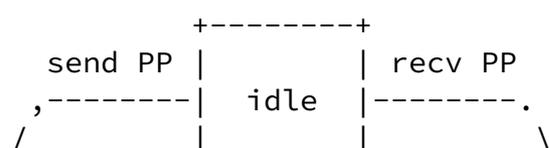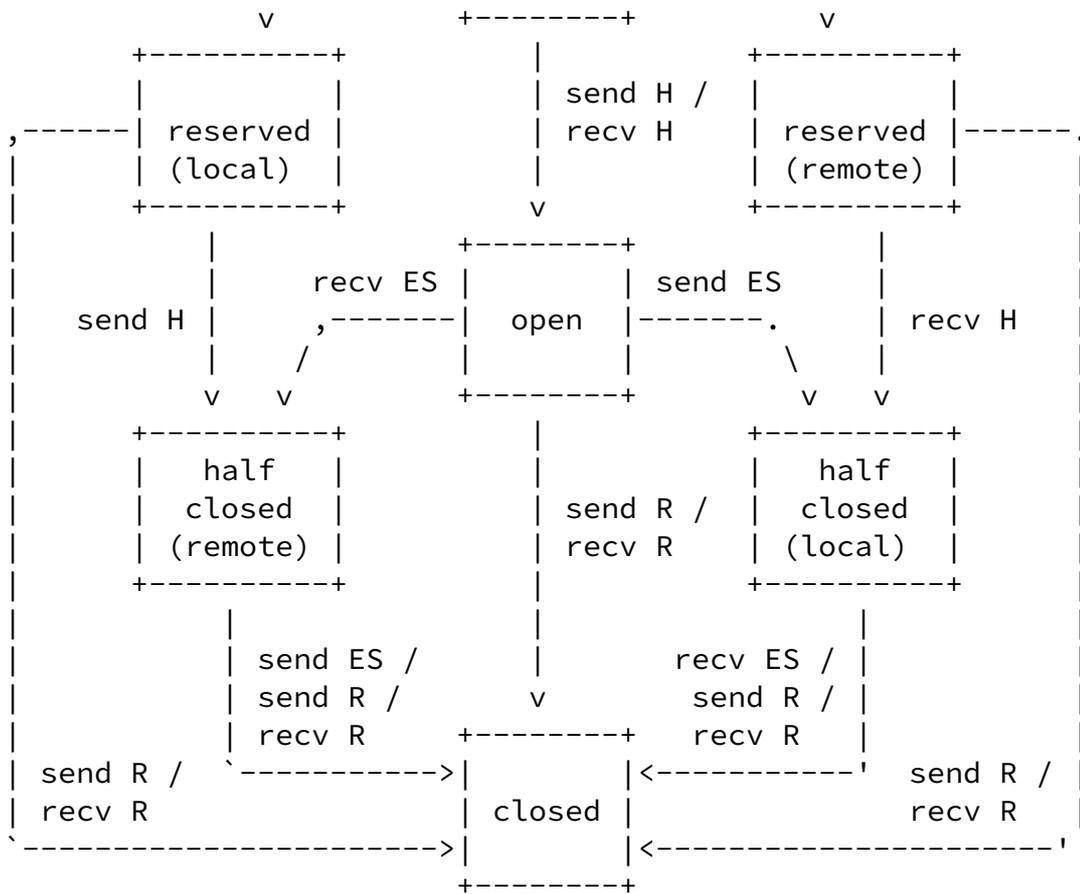
```
      send:   endpoint sends this frame
      recv:   endpoint receives this frame

      H:  HEADERS frame (with implied CONTINUATIONs)
      ES: END_STREAM flag
      R:  RST_STREAM frame
      PP: PUSH_PROMISE frame (with implied CONTINUATIONs)
          Note: State transitions are for the promised stream.
```

                    Figure 2: Stream States

   Note that this diagram shows stream state transitions and the frames
   and flags that affect those transitions only. In this regard,
   CONTINUATION frames do not result in state transitions; they are
   effectively part of the HEADERS or PUSH_PROMISE that they follow.

For the purpose of state transitions, the END_STREAM flag is
processed as a separate event to the frame that bears it; a HEADERS
frame with the END_STREAM flag set can cause two state transitions.

Both endpoints have a subjective view of the state of a stream that
could be different when frames are in transit. Endpoints do not
coordinate the creation of streams; they are created unilaterally by
either endpoint. The negative consequences of a mismatch in states
are limited to the "closed" state after sending RST_STREAM, where
frames might be received for some time after closing.

Streams have the following states:

idle: All streams start in the "idle" state.

   The following transitions are valid from this state:

   * Sending or receiving a HEADERS frame causes the stream to
      become "open". The stream identifier is selected as described
      in Section 5.1.1. The same HEADERS frame can also cause a
      stream to immediately become "half-closed".

   * Sending a PUSH_PROMISE frame on another stream reserves the
      idle stream that is identified for later use. The stream state
      for the reserved stream transitions to "reserved (local)".

   * Receiving a PUSH_PROMISE frame on another stream reserves an
      idle stream that is identified for later use. The stream state
      for the reserved stream transitions to "reserved (remote)".

   * Note that the PUSH_PROMISE frame is not sent on the idle stream
      but references the newly reserved stream in the Promised Stream
      ID field.

   Receiving any frame other than HEADERS or PRIORITY on a stream in
   this state MUST be treated as a connection error (Section 5.4.1)
   of type PROTOCOL_ERROR.

reserved (local): A stream in the "reserved (local)" state is one
   that has been promised by sending a PUSH_PROMISE frame. A
   PUSH_PROMISE frame reserves an idle stream by associating the
   stream with an open stream that was initiated by the remote peer
   (see Section 8.2).

In this state, only the following transitions are possible:

* The endpoint can send a HEADERS frame. This causes the stream
  to open in a "half-closed (remote)" state.

* Either endpoint can send a RST_STREAM frame to cause the stream
  to become "closed". This releases the stream reservation.

An endpoint MUST NOT send any type of frame other than HEADERS,
RST_STREAM, or PRIORITY in this state.

A PRIORITY or WINDOW_UPDATE frame MAY be received in this state.
Receiving any type of frame other than RST_STREAM, PRIORITY, or
WINDOW_UPDATE on a stream in this state MUST be treated as a
connection error (Section 5.4.1) of type PROTOCOL_ERROR.

reserved (remote): A stream in the "reserved (remote)" state has
   been reserved by a remote peer.

   In this state, only the following transitions are possible:

   * Receiving a HEADERS frame causes the stream to transition to
     "half-closed (local)".

   * Either endpoint can send a RST_STREAM frame to cause the stream
     to become "closed". This releases the stream reservation.

   An endpoint MUST NOT send any type of frame other than RST_STREAM,
   WINDOW_UPDATE, or PRIORITY in this state.

   Receiving any type of frame other than HEADERS, RST_STREAM, or
   PRIORITY on a stream in this state MUST be treated as a connection
   error (Section 5.4.1) of type PROTOCOL_ERROR.

open: A stream in the "open" state may be used by both peers to send
   frames of any type. In this state, sending peers observe
   advertised stream-level flow-control limits (Section 5.2).

   From this state, either endpoint can send a frame with an
   END_STREAM flag set, which causes the stream to transition into
   one of the "half-closed" states. An endpoint sending an
   END_STREAM flag causes the stream state to become "half-closed
   (local)"; an endpoint receiving an END_STREAM flag causes the
   stream state to become "half-closed (remote)".

   Either endpoint can send a RST_STREAM frame from this state,
   causing it to transition immediately to "closed".

half-closed (local): A stream that is in the "half-closed (local)"
   state cannot be used for sending frames other than WINDOW_UPDATE,
   PRIORITY, and RST_STREAM.

   A stream transitions from this state to "closed" when a frame that
   contains an END_STREAM flag is received or when either peer sends
   a RST_STREAM frame.

   An endpoint can receive any type of frame in this state.
   Providing flow-control credit using WINDOW_UPDATE frames is
   necessary to continue receiving flow-controlled frames. In this
   state, a receiver can ignore WINDOW_UPDATE frames, which might
   arrive for a short period after a frame bearing the END_STREAM
   flag is sent.

   PRIORITY frames can be received in this state.

half-closed (remote): A stream that is "half-closed (remote)" is no
   longer being used by the peer to send frames. In this state, an
   endpoint is no longer obligated to maintain a receiver flow-
   control window.

   If an endpoint receives additional frames, other than
   WINDOW_UPDATE, PRIORITY, or RST_STREAM, for a stream that is in
   this state, it MUST respond with a stream error ([Section 5.4.2](#)) of
   type STREAM_CLOSED.

   A stream that is "half-closed (remote)" can be used by the
   endpoint to send frames of any type. In this state, the endpoint
   continues to observe advertised stream-level flow-control limits
   ([Section 5.2](#)).

   A stream can transition from this state to "closed" by sending a
   frame that contains an END_STREAM flag or when either peer sends a
   RST_STREAM frame.

closed: The "closed" state is the terminal state.

   An stream enters the "closed" state after an endpoint both sends
   and receives a frame with an END_STREAM flag set. A stream also

enters the "closed" state after an endpoint either sends or
receives a RST_STREAM frame.

An endpoint MUST NOT send frames other than PRIORITY on a closed
stream. An endpoint MAY treat receipt of any other type of frame
on a "closed" stream as a connection error ([Section 5.4.1](#)) of type
STREAM_CLOSED, except as noted below.

An endpoint that sends a frame with the END_STREAM flag set or a
RST_STREAM frame might receive a WINDOW_UPDATE or RST_STREAM frame
from its peer in the time before the peer receives and processes
the frame that closes the stream.

An endpoint that sends a RST_STREAM frame on a stream that is in
the "open" state could receive any type of frame. The peer might
have sent or enqueued for sending these frames before processing
the RST_STREAM frame. An endpoint MUST minimally process and then
discard any frames it receives in this state. This means updating
header compression state for HEADERS and PUSH_PROMISE frames;
PUSH_PROMISE frames also cause the promised stream to become
"reserved", even when the PUSH_PROMISE frame is received on a
closed stream; and, the content of DATA frames counts toward the
connection flow-control window.

An endpoint can perform this minimal processing for all streams
that are in the "closed" state. Endpoints MAY use other signals
to detect that a peer has received the frames that caused stream
to become "closed" and treat receipt of any frame other than
PRIORITY as a connection error ([Section 5.4.1](#)) of type
PROTOCOL_ERROR. Endpoints can use frames that indicate that the
peer has received the closing signal to drive this. Endpoints
SHOULD NOT use timers for this purpose. For example, an endpoint
that sends a SETTINGS frame after closing a stream can safely
treat receipt of a DATA frame on that stream as an error after
receiving an acknowledgement of the settings. Other things that
might be used are PING frames, receiving data on streams that were
created after closing the stream, or responses to requests created
after closing the stream.

In the absence of more specific guidance elsewhere in this document,
implementations SHOULD treat the receipt of a frame that is not

expressly permitted in the description of a state as a connection error ([Section 5.4.1](#)) of type PROTOCOL_ERROR. Note that PRIORITY can be sent and received in any stream state. Frames of unknown types are ignored.

An example of the state transitions for an HTTP request/response exchange can be found in [Section 8.1](#). An example of the state transitions for server push can be found in Sections [8.2.1](#) and [8.2.2](#).

[5.1.1](#). Stream Identifiers

Streams are identified with an unsigned 31-bit integer. Streams initiated by a client MUST use odd-numbered stream identifiers; those initiated by the server MUST use even-numbered stream identifiers. A stream identifier of zero (0x0) is used for connection control messages; the stream identifier of zero cannot be used to establish a new stream.

The identifier of a newly established stream MUST be numerically greater than all streams that the initiating endpoint has opened or reserved. This governs streams that are opened using a HEADERS frame and streams that are reserved using PUSH_PROMISE. An endpoint that receives an unexpected stream identifier MUST respond with a connection error ([Section 5.4.1](#)) of type PROTOCOL_ERROR.

A HEADERS frame will transition the client-initiated stream identified by the stream identifier in the frame header from "idle" to "open". A PUSH_PROMISE frame will transition the server-initiated stream identified by the "Promised Stream ID" field in the frame payload from "idle" to "reserved". When a stream transitions out of the "idle" state, all streams that might have been initiated by that peer with a lower-valued stream identifier are implicitly transitioned to "closed". That is, an endpoint may skip a stream identifier, with the effect being that the skipped stream is immediately closed.

Stream identifiers cannot be reused. Long-lived connections can
result in an endpoint exhausting the available range of stream
identifiers. A client that is unable to establish a new stream
identifier can establish a new connection for new streams. A server
that is unable to establish a new stream identifier can send a GOAWAY
frame so that the client is forced to open a new connection for new
streams.

5.1.2. Stream Concurrency

A peer can limit the number of concurrently active streams using the
SETTINGS_MAX_CONCURRENT_STREAMS parameter (see Section 6.5.2) within
a SETTINGS frame. The maximum concurrent streams setting is specific
to each endpoint and applies only to the peer that receives the
setting. That is, clients specify the maximum number of concurrent
streams the server can initiate, and servers specify the maximum
number of concurrent streams the client can initiate.

Streams that are in the "open" state or in either of the "half-
closed" states count toward the maximum number of streams that an
endpoint is permitted to open. Streams in any of these three states

count toward the limit advertised in the
SETTINGS_MAX_CONCURRENT_STREAMS setting. Streams in either of the
"reserved" states do not count toward the stream limit.

Endpoints MUST NOT exceed the limit set by their peer. An endpoint
that receives a HEADERS frame that causes its advertised concurrent
stream limit to be exceeded MUST treat this as a stream error
(Section 5.4.2) of type PROTOCOL_ERROR or REFUSED_STREAM. The choice
of error code determines whether the endpoint wishes to enable
automatic retry (see Section 8.1.4) for details).

An endpoint that wishes to reduce the value of
SETTINGS_MAX_CONCURRENT_STREAMS to a value that is below the current
number of open streams can either close streams that exceed the new
value or allow streams to complete.

5.2. Flow Control

Using streams for multiplexing introduces contention over use of the

TCP connection, resulting in blocked streams. A flow-control scheme
ensures that streams on the same connection do not destructively
interfere with each other. Flow control is used for both individual
streams and for the connection as a whole.

HTTP/2 provides for flow control through use of the WINDOW_UPDATE
frame (Section 6.9).

5.2.1. Flow-Control Principles

HTTP/2 stream flow control aims to allow a variety of flow-control
algorithms to be used without requiring protocol changes. Flow
control in HTTP/2 has the following characteristics:

1. Flow control is specific to a connection. Both types of flow
    control are between the endpoints of a single hop and not over
    the entire end-to-end path.

2. Flow control is based on WINDOW_UPDATE frames. Receivers
    advertise how many octets they are prepared to receive on a
    stream and for the entire connection. This is a credit-based
    scheme.

3. Flow control is directional with overall control provided by the
    receiver. A receiver MAY choose to set any window size that it
    desires for each stream and for the entire connection. A sender
    MUST respect flow-control limits imposed by a receiver. Clients,
    servers, and intermediaries all independently advertise their
    flow-control window as a receiver and abide by the flow-control
    limits set by their peer when sending.

4. The initial value for the flow-control window is 65,535 octets
    for both new streams and the overall connection.

5. The frame type determines whether flow control applies to a

frame. Of the frames specified in this document, only DATA
      frames are subject to flow control; all other frame types do not
      consume space in the advertised flow-control window. This
      ensures that important control frames are not blocked by flow
      control.

   6. Flow control cannot be disabled.

   7. HTTP/2 defines only the format and semantics of the WINDOW_UPDATE
       frame (Section 6.9). This document does not stipulate how a
       receiver decides when to send this frame or the value that it
       sends, nor does it specify how a sender chooses to send packets.
       Implementations are able to select any algorithm that suits their
       needs.

   Implementations are also responsible for prioritizing the sending of
   requests and responses, choosing how to avoid head-of-line blocking
   for requests, and managing the creation of new streams. Algorithm
   choices for these could interact with any flow-control algorithm.

5.2.2. Appropriate Use of Flow Control

   Flow control is defined to protect endpoints that are operating under
   resource constraints. For example, a proxy needs to share memory
   between many connections and also might have a slow upstream
   connection and a fast downstream one. Flow-control addresses cases
   where the receiver is unable to process data on one stream yet wants
   to continue to process other streams in the same connection.

   Deployments that do not require this capability can advertise a flow-
   control window of the maximum size ($2^{31}-1$) and can maintain this
   window by sending a WINDOW_UPDATE frame when any data is received.
   This effectively disables flow control for that receiver.
   Conversely, a sender is always subject to the flow-control window
   advertised by the receiver.

   Deployments with constrained resources (for example, memory) can
   employ flow control to limit the amount of memory a peer can consume.
   Note, however, that this can lead to suboptimal use of available
   network resources if flow control is enabled without knowledge of the
   bandwidth-delay product (see [RFC7323]).

Even with full awareness of the current bandwidth-delay product, implementation of flow control can be difficult. When using flow control, the receiver MUST read from the TCP receive buffer in a timely fashion. Failure to do so could lead to a deadlock when critical frames, such as WINDOW_UPDATE, are not read and acted upon.

5.2.3. Flow Control Performance

If an endpoint cannot ensure that its peer always has available flow control window space that is greater than the peer's bandwidth-delay product on this connection, its receive throughput will be limited by HTTP/2 flow control. This will result in degraded performance.

Sending timely WINDOW_UPDATE frames can improve performance. Endpoints will want to balance the need to improve receive throughput with the need to manage resource exhaustion risks, and should take careful note of Section 10.5 in defining their strategy to manage window sizes.

5.3. Prioritization

In a multiplexed protocol like HTTP/2, prioritizing allocation of bandwidth and computation resources to streams can be critical to attaining good performance. A poor prioritization scheme can result in HTTP/2 providing poor performance. With no parallelism at the TCP layer, performance could be significantly worse than HTTP/1.1.

A good prioritization scheme benefits from the application of contextual knowledge such as the content of resources, how resources are interrelated, and how those resources will be used by a peer. In particular, clients can possess knowledge about the priority of requests that is relevant to server prioritization. In those cases, having clients provide priority information can improve performance.

5.3.1. Background of Priority in HTTP/2

HTTP/2 included a rich system for signaling priority of requests. However, this system proved to be complex and it was not uniformly implemented.

The flexible scheme meant that it was possible for clients to express priorities in very different ways, with little consistency in the approaches that were adopted. For servers, implementing generic support for the scheme was complex. Implementation of priorities was uneven in both clients and servers. Many server deployments ignored client signals when prioritizing their handling of requests.

In short, the prioritization signaling in RFC7540 [RFC7540] was not successful.

5.3.2. Priority Signaling in this Document

This update to HTTP/2 deprecates the priority signaling defined in RFC 7540 [RFC7540]. The bulk of the text related to priority signals is not included in this document. The description of frame fields and some of the mandatory handling is retained to ensure that implementations of this document remain interoperable with implementations that use the priority signaling described in RFC 7540.

A thorough description of the RFC 7540 priority scheme remains in Section 5.3 of [RFC7540].

Signaling priority information is necessary to attain good performance in many cases. Where signaling priority information is important, endpoints are encouraged to use an alternative scheme, such as [I-D.ietf-httpbis-priority].

Though the priority signaling from RFC 7540 was not widely adopted, the information it provides can still be useful in the absence of better information. Endpoints that receive priority signals in HEADERS or PRIORITY frames can benefit from applying that information. In particular, implementations that consume these signals would not benefit from discarding these priority signals in the absence of alternatives.

Servers SHOULD use other contextual information in determining priority of requests in the absence of any priority signals. Servers MAY interpret the complete absence of signals as an indication that the client has not implemented the feature. The defaults described in Section 5.3.5 of [RFC7540] are known to have poor performance under most conditions and their use is unlikely to be deliberate.

5.4. Error Handling

HTTP/2 framing permits two classes of error:

   * An error condition that renders the entire connection unusable is
      a connection error.

   * An error in an individual stream is a stream error.

   A list of error codes is included in [Section 7](#).

   It is possible that an endpoint will encounter frames that would
   cause multiple errors. Implementations MAY discover multiple errors
   during processing, but they SHOULD report at most one stream and one
   connection error as a result.

   The first stream error reported for a given stream prevents any other
   errors on that stream from being reported. In comparison, the
   protocol permits multiple GOAWAY frames, though an endpoint SHOULD
   report just one type of connection error unless an error is
   encountered during graceful shutdown. If this occurs, an endpoint
   MAY send an additional GOAWAY frame with the new error code, in
   addition to any prior GOAWAY that contained NO_ERROR.

   If an endpoint detects multiple different errors, it MAY choose to
   report any one of those errors. If a frame causes a connection
   error, that error MUST be reported. Additionally, an endpoint MAY
   use any applicable error code when it detects an error condition; a
   generic error code (such as PROTOCOL_ERROR or INTERNAL_ERROR) can
   always be used in place of more specific error codes.

5.4.1. Connection Error Handling

   A connection error is any error that prevents further processing of
   the frame layer or corrupts any connection state.

   An endpoint that encounters a connection error SHOULD first send a
   GOAWAY frame ([Section 6.8](#)) with the stream identifier of the last
   stream that it successfully received from its peer. The GOAWAY frame
   includes an error code that indicates why the connection is
   terminating. After sending the GOAWAY frame for an error condition,
   the endpoint MUST close the TCP connection.

   It is possible that the GOAWAY will not be reliably received by the
   receiving endpoint. In the event of a connection error, GOAWAY only
   provides a best-effort attempt to communicate with the peer about why

the connection is being terminated.

An endpoint can end a connection at any time. In particular, an
endpoint MAY choose to treat a stream error as a connection error.
Endpoints SHOULD send a GOAWAY frame when ending a connection,
providing that circumstances permit it.

5.4.2. Stream Error Handling

A stream error is an error related to a specific stream that does not
affect processing of other streams.

An endpoint that detects a stream error sends a RST_STREAM frame
(Section 6.4) that contains the stream identifier of the stream where
the error occurred. The RST_STREAM frame includes an error code that
indicates the type of error.

A RST_STREAM is the last frame that an endpoint can send on a stream.
The peer that sends the RST_STREAM frame MUST be prepared to receive
any frames that were sent or enqueued for sending by the remote peer.
These frames can be ignored, except where they modify connection
state (such as the state maintained for field section compression
(Section 4.3) or flow control).

Normally, an endpoint SHOULD NOT send more than one RST_STREAM frame
for any stream. However, an endpoint MAY send additional RST_STREAM
frames if it receives frames on a closed stream after more than a
round-trip time. This behavior is permitted to deal with misbehaving
implementations.

To avoid looping, an endpoint MUST NOT send a RST_STREAM in response
to a RST_STREAM frame.

5.4.3. Connection Termination

If the TCP connection is closed or reset while streams remain in
"open" or "half-closed" state, then the affected streams cannot be
automatically retried (see Section 8.1.4 for details).

5.5. Extending HTTP/2

HTTP/2 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/2 connection.

This applies to the protocol elements defined in this document. This
does not affect the existing options for extending HTTP, such as
defining new methods, status codes, or fields (see Section 16 of
[HTTP]).

Extensions are permitted to use new frame types (Section 4.1), new
settings (Section 6.5.2), or new error codes (Section 7). Registries
are established for managing these extension points: frame types
(Section 11.2), settings (Section 11.3), and error codes
(Section 11.4).

Implementations MUST ignore unknown or unsupported values in all
extensible protocol elements. Implementations MUST discard frames
that have unknown or unsupported types. This means that any of these
extension points can be safely used by extensions without prior
arrangement or negotiation. However, extension frames that appear in
the middle of a field block (Section 4.3) are not permitted; these
MUST be treated as a connection error (Section 5.4.1) of type
PROTOCOL_ERROR.

Extensions SHOULD avoiding changing protocol elements defined in this
document or elements for which no extension mechanism is defined.
This includes changes to the layout of frames, additions or changes
to the way that frames are composed into HTTP messages (Section 8),
the definition of pseudo-header fields, or changes to any protocol
element that a compliant endpoint might treat as a connection error
(Section 5.4.1).

An extension that changes existing elements 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.

For example, treating frames other than DATA frames as flow
controlled requires a change in semantics that both endpoints need to
understand, so this can only be done through negotiation.

This document doesn't mandate a specific method for negotiating the
use of an extension but notes that a setting (Section 6.5.2) 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 initial value MUST
be defined in such a fashion that the extension is initially
disabled.

6. Frame Definitions

This specification defines a number of frame types, each identified
by a unique 8-bit type code. Each frame type serves a distinct
purpose in the establishment and management either of the connection
as a whole or of individual streams.

The transmission of specific frame types can alter the state of a
connection. If endpoints fail to maintain a synchronized view of the
connection state, successful communication within the connection will
no longer be possible. Therefore, it is important that endpoints
have a shared comprehension of how the state is affected by the use
any given frame.

6.1. DATA

DATA frames (type=0x0) convey arbitrary, variable-length sequences of
octets associated with a stream. One or more DATA frames are used,
for instance, to carry HTTP request or response message contents.

DATA frames MAY also contain padding. Padding can be added to DATA
frames to obscure the size of messages. Padding is a security
feature; see Section 10.7.

```
  +---------------+
  |Pad Length? (8)|
  +---------------+-----------------------------------------------+
  |                            Data (*)                         ...
```

```
+---------------------------------------------------------------+
|                            Padding (*)                    ...
+---------------------------------------------------------------+
```

                        Figure 3: DATA Frame Payload

   The DATA frame contains the following fields:

   Pad Length: An 8-bit field containing the length of the frame
      padding in units of octets. This field is conditional (as
      signified by a "?" in the diagram) and is only present if the
      PADDED flag is set.

   Data: Application data. The amount of data is the remainder of the
      frame payload after subtracting the length of the other fields
      that are present.

   Padding: Padding octets that contain no application semantic value.
      Padding octets MUST be set to zero when sending. A receiver is
      not obligated to verify padding but MAY treat non-zero padding as
      a connection error ([Section 5.4.1](#)) of type PROTOCOL_ERROR.

   The DATA frame defines the following flags:

   END_STREAM (0x1): When set, bit 0 indicates that this frame is the

      last that the endpoint will send for the identified stream.
      Setting this flag causes the stream to enter one of the
      "half-closed" states or the "closed" state ([Section 5.1](#)).

   PADDED (0x8): When set, bit 3 indicates that the Pad Length field
      and any padding that it describes are present.

   DATA frames MUST be associated with a stream. If a DATA frame is
   received whose stream identifier field is 0x0, the recipient MUST
   respond with a connection error ([Section 5.4.1](#)) of type
   PROTOCOL_ERROR.

   DATA frames are subject to flow control and can only be sent when a
   stream is in the "open" or "half-closed (remote)" state. The entire

DATA frame payload is included in flow control, including the Pad
Length and Padding fields if present. If a DATA frame is received
whose stream is not in "open" or "half-closed (local)" state, the
recipient MUST respond with a stream error ([Section 5.4.2](#)) of type
STREAM_CLOSED.

The total number of padding octets is determined by the value of the
Pad Length field. If the length of the padding is the length of the
frame payload or greater, the recipient MUST treat this as a
connection error ([Section 5.4.1](#)) of type PROTOCOL_ERROR.

> | Note: A frame can be increased in size by one octet by
> | including a Pad Length field with a value of zero.

[6.2](#). HEADERS

The HEADERS frame (type=0x1) is used to open a stream ([Section 5.1](#)),
and additionally carries a field block fragment. Despite the name, a
HEADERS frame can carry a header section or a trailer section.
HEADERS frames can be sent on a stream in the "idle", "reserved
(local)", "open", or "half-closed (remote)" state.

```
+---------------+
|Pad Length? (8)|
+-+-------------+-----------------------------------------------+
|E|                 Stream Dependency? (31)                     |
+-+-------------+-----------------------------------------------+
|  Weight? (8)  |
+-+-------------+-----------------------------------------------+
|                   Field Block Fragment (*)                ...
+---------------------------------------------------------------+
|                          Padding (*)                      ...
+---------------------------------------------------------------+
```
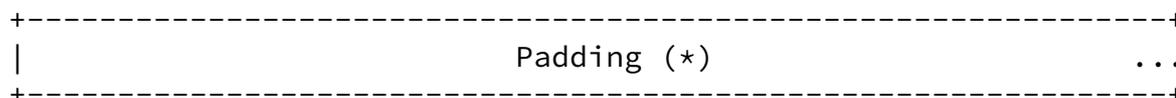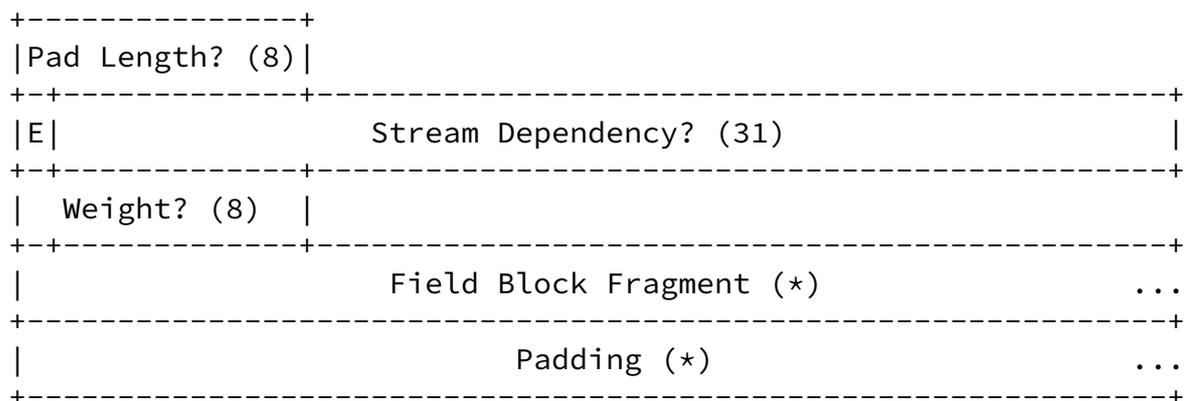
                    Figure 4: HEADERS Frame Payload

The HEADERS frame payload has the following fields:

Pad Length: An 8-bit field containing the length of the frame
   padding in units of octets. This field is only present if the
   PADDED flag is set.

E: A single-bit flag. This field is only present if the PRIORITY
    flag is set.

Stream Dependency: A 31-bit stream identifier. This field is only
    present if the PRIORITY flag is set.

Weight: An unsigned 8-bit integer. This field is only present if
    the PRIORITY flag is set.

Field Block Fragment: A field block fragment (Section 4.3).

Padding: Padding octets.

The HEADERS frame defines the following flags:

END_STREAM (0x1): When set, bit 0 indicates that the field block
    (Section 4.3) is the last that the endpoint will send for the
    identified stream.

    A HEADERS frame carries the END_STREAM flag that signals the end
    of a stream. However, a HEADERS frame with the END_STREAM flag
    set can be followed by CONTINUATION frames on the same stream.
    Logically, the CONTINUATION frames are part of the HEADERS frame.

END_HEADERS (0x4): When set, bit 2 indicates that this frame
    contains an entire field block (Section 4.3) and is not followed
    by any CONTINUATION frames.

    A HEADERS frame without the END_HEADERS flag set MUST be followed
    by a CONTINUATION frame for the same stream. A receiver MUST
    treat the receipt of any other type of frame or a frame on a
    different stream as a connection error (Section 5.4.1) of type
    PROTOCOL_ERROR.

PADDED (0x8): When set, bit 3 indicates that the Pad Length field
    and any padding that it describes are present.

PRIORITY (0x20): When set, bit 5 indicates that the Exclusive Flag
    (E), Stream Dependency, and Weight fields are present.

The frame payload of a HEADERS frame contains a field block fragment (Section 4.3). A field block that does not fit within a HEADERS frame is continued in a CONTINUATION frame (Section 6.10).

HEADERS frames MUST be associated with a stream. If a HEADERS frame is received whose stream identifier field is 0x0, the recipient MUST respond with a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

The HEADERS frame changes the connection state as described in Section 4.3.

The HEADERS frame can include padding. Padding fields and flags are identical to those defined for DATA frames (Section 6.1). Padding that exceeds the size remaining for the field block fragment MUST be treated as a PROTOCOL_ERROR.

6.3. PRIORITY

The PRIORITY frame (type=0x2) is deprecated; see Section 5.3.2. A PRIORITY frame can be sent in any stream state, including idle or closed streams.

```
+-+-------------------------------------------------------------+
|E|                  Stream Dependency (31)                     |
+-+-------------+-----------------------------------------------+
|   Weight (8)  |
+-+-------------+
```
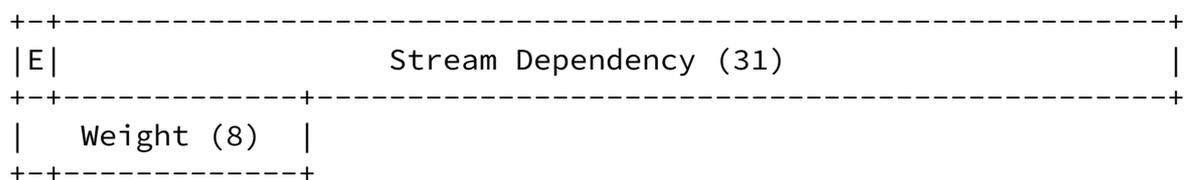
                   Figure 5: PRIORITY Frame Payload

The frame payload of a PRIORITY frame contains the following fields:

E: A single-bit flag.

Stream Dependency: A 31-bit stream identifier.

Weight: An unsigned 8-bit integer.

The PRIORITY frame does not define any flags.

The PRIORITY frame always identifies a stream. If a PRIORITY frame is received with a stream identifier of 0x0, the recipient MUST respond with a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

Sending or receiving a PRIORITY frame does not affect the state of
any stream (Section 5.1). The PRIORITY frame can be sent on a stream
in any state, including "idle" or "closed". A PRIORITY frame cannot
be sent between consecutive frames that comprise a single field block
(Section 4.3).

A PRIORITY frame with a length other than 5 octets MUST be treated as
a stream error (Section 5.4.2) of type FRAME_SIZE_ERROR.

6.4. RST_STREAM

The RST_STREAM frame (type=0x3) allows for immediate termination of a
stream. RST_STREAM is sent to request cancellation of a stream or to
indicate that an error condition has occurred.

```
+-------------------------------------------------------------------+
|                         Error Code (32)                           |
+-------------------------------------------------------------------+
```

                   Figure 6: RST_STREAM Frame Payload

The RST_STREAM frame contains a single unsigned, 32-bit integer
identifying the error code (Section 7). The error code indicates why
the stream is being terminated.

The RST_STREAM frame does not define any flags.

The RST_STREAM frame fully terminates the referenced stream and
causes it to enter the "closed" state. After receiving a RST_STREAM
on a stream, the receiver MUST NOT send additional frames for that
stream, with the exception of PRIORITY. However, after sending the
RST_STREAM, the sending endpoint MUST be prepared to receive and
process additional frames sent on the stream that might have been
sent by the peer prior to the arrival of the RST_STREAM.

RST_STREAM frames MUST be associated with a stream. If a RST_STREAM
frame is received with a stream identifier of 0x0, the recipient MUST
treat this as a connection error (Section 5.4.1) of type
PROTOCOL_ERROR.

RST_STREAM frames MUST NOT be sent for a stream in the "idle" state.
If a RST_STREAM frame identifying an idle stream is received, the
recipient MUST treat this as a connection error (Section 5.4.1) of

type PROTOCOL_ERROR.

A RST_STREAM frame with a length other than 4 octets MUST be treated
as a connection error (Section 5.4.1) of type FRAME_SIZE_ERROR.

6.5. SETTINGS

The SETTINGS frame (type=0x4) conveys configuration parameters that
affect how endpoints communicate, such as preferences and constraints
on peer behavior. The SETTINGS frame is also used to acknowledge the
receipt of those settings. Individually, a configuration parameter
from a SETTINGS frame is referred to as a "setting".

Settings are not negotiated; they describe characteristics of the
sending peer, which are used by the receiving peer. Different values
for the same setting can be advertised by each peer. For example, a
client might set a high initial flow-control window, whereas a server
might set a lower value to conserve resources.

A SETTINGS frame MUST be sent by both endpoints at the start of a
connection and MAY be sent at any other time by either endpoint over
the lifetime of the connection. Implementations MUST support all of
the settings defined by this specification.

Each parameter in a SETTINGS frame replaces any existing value for
that parameter. Settings are processed in the order in which they
appear, and a receiver of a SETTINGS frame does not need to maintain
any state other than the current value of each setting. Therefore,
the value of a SETTINGS parameter is the last value that is seen by a
receiver.

SETTINGS frames are acknowledged by the receiving peer. To enable
this, the SETTINGS frame defines the ACK flag:

ACK (0x1): When set, bit 0 indicates that this frame acknowledges
   receipt and application of the peer's SETTINGS frame. When this
   bit is set, the frame payload of the SETTINGS frame MUST be empty.
   Receipt of a SETTINGS frame with the ACK flag set and a length
   field value other than 0 MUST be treated as a connection error
   (Section 5.4.1) of type FRAME_SIZE_ERROR. For more information,
   see Section 6.5.3 ("Settings Synchronization").

SETTINGS frames always apply to a connection, never a single stream.
The stream identifier for a SETTINGS frame MUST be zero (0x0). If an
endpoint receives a SETTINGS frame whose stream identifier field is
anything other than 0x0, the endpoint MUST respond with a connection
error ([Section 5.4.1](#)) of type PROTOCOL_ERROR.

The SETTINGS frame affects connection state. A badly formed or
incomplete SETTINGS frame MUST be treated as a connection error
([Section 5.4.1](#)) of type PROTOCOL_ERROR.

A SETTINGS frame with a length other than a multiple of 6 octets MUST
be treated as a connection error ([Section 5.4.1](#)) of type
FRAME_SIZE_ERROR.

6.5.1. SETTINGS Format

The frame payload of a SETTINGS frame consists of zero or more
settings, each consisting of an unsigned 16-bit setting identifier
and an unsigned 32-bit value.

```
    +-------------------------------+
    |         Identifier (16)       |
    +-------------------------------+-------------------------------+
    |                        Value (32)                             |
    +---------------------------------------------------------------+
```
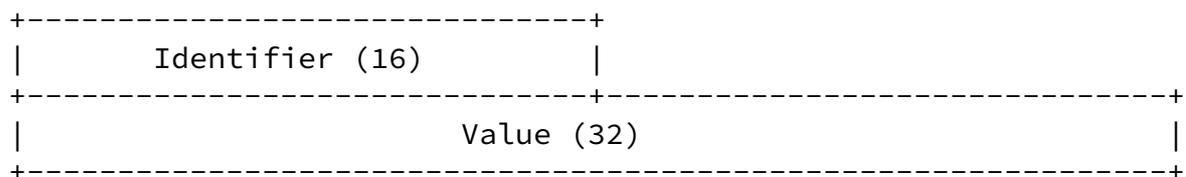
Figure 7: Setting Format

6.5.2. Defined Settings

The following settings are defined:

SETTINGS_HEADER_TABLE_SIZE (0x1): Allows the sender to inform the
   remote endpoint of the maximum size of the compression table used
   to decode field blocks, in octets. The encoder can select any
   size equal to or less than this value by using signaling specific
   to the compression format inside a field block (see
   [COMPRESSION]). The initial value is 4,096 octets.

SETTINGS_ENABLE_PUSH (0x2): This setting can be used to disable
   server push (Section 8.2). A server MUST NOT send a PUSH_PROMISE
   frame if it receives this parameter set to a value of 0. A client
   that has both set this parameter to 0 and had it acknowledged MUST
   treat the receipt of a PUSH_PROMISE frame as a connection error
   (Section 5.4.1) of type PROTOCOL_ERROR.

   The initial value of SETTINGS_ENABLE_PUSH is 1, which indicates
   that server push is permitted. Any value other than 0 or 1 MUST
   be treated as a connection error (Section 5.4.1) of type
   PROTOCOL_ERROR.

   A server MUST NOT explicitly set this value to 1. A server MAY
   choose to omit this setting when it sends a SETTINGS frame, but if
   a server does include a value it MUST be 0. A client MUST treat
   receipt of a SETTINGS frame with SETTINGS_ENABLE_PUSH set to 1 as
   a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

SETTINGS_MAX_CONCURRENT_STREAMS (0x3): Indicates the maximum number

   of concurrent streams that the sender will allow. This limit is
   directional: it applies to the number of streams that the sender
   permits the receiver to create. Initially, there is no limit to
   this value. It is recommended that this value be no smaller than
   100, so as to not unnecessarily limit parallelism.

   A value of 0 for SETTINGS_MAX_CONCURRENT_STREAMS SHOULD NOT be
   treated as special by endpoints. A zero value does prevent the
   creation of new streams; however, this can also happen for any
   limit that is exhausted with active streams. Servers SHOULD only
   set a zero value for short durations; if a server does not wish to
   accept requests, closing the connection is more appropriate.

SETTINGS_INITIAL_WINDOW_SIZE (0x4): Indicates the sender's initial
   window size (in octets) for stream-level flow control. The
   initial value is 2^16-1 (65,535) octets.

   This setting affects the window size of all streams (see
   Section 6.9.2).

   Values above the maximum flow-control window size of 2^31-1 MUST
   be treated as a connection error (Section 5.4.1) of type

FLOW_CONTROL_ERROR.

   SETTINGS_MAX_FRAME_SIZE (0x5): Indicates the size of the largest
      frame payload that the sender is willing to receive, in octets.

      The initial value is 2^14 (16,384) octets. The value advertised
      by an endpoint MUST be between this initial value and the maximum
      allowed frame size (2^24-1 or 16,777,215 octets), inclusive.
      Values outside this range MUST be treated as a connection error
      ([Section 5.4.1](#)) of type PROTOCOL_ERROR.

   SETTINGS_MAX_HEADER_LIST_SIZE (0x6): This advisory setting informs a
      peer of the maximum size of field section that the sender is
      prepared to accept, in octets. The value is based on the
      uncompressed size of field lines, including the length of the name
      and value in octets plus an overhead of 32 octets for each field
      line.

      For any given request, a lower limit than what is advertised MAY
      be enforced. The initial value of this setting is unlimited.

   An endpoint that receives a SETTINGS frame with any unknown or
   unsupported identifier MUST ignore that setting.

[6.5.3](#). Settings Synchronization

   Most values in SETTINGS benefit from or require an understanding of
   when the peer has received and applied the changed parameter values.
   In order to provide such synchronization timepoints, the recipient of
   a SETTINGS frame in which the ACK flag is not set MUST apply the
   updated settings as soon as possible upon receipt.

   The values in the SETTINGS frame MUST be processed in the order they
   appear, with no other frame processing between values. Unsupported
   settings MUST be ignored. Once all values have been processed, the
   recipient MUST immediately emit a SETTINGS frame with the ACK flag
   set. Upon receiving a SETTINGS frame with the ACK flag set, the
   sender of the altered settings can rely on the value having been
   applied.

If the sender of a SETTINGS frame does not receive an acknowledgement within a reasonable amount of time, it MAY issue a connection error (Section 5.4.1) of type SETTINGS_TIMEOUT.

6.6. PUSH_PROMISE

The PUSH_PROMISE frame (type=0x5) is used to notify the peer endpoint in advance of streams the sender intends to initiate. The PUSH_PROMISE frame includes the unsigned 31-bit identifier of the stream the endpoint plans to create along with a field section that provides additional context for the stream. Section 8.2 contains a thorough description of the use of PUSH_PROMISE frames.

```
 +---------------+
 |Pad Length? (8)|
 +-+-------------+-----------------------------------------------+
 |R|                  Promised Stream ID (31)                    |
 +-+-----------------------------+-------------------------------+
 |                   Field Block Fragment (*)                  ...
 +---------------------------------------------------------------+
 |                         Padding (*)                         ...
 +---------------------------------------------------------------+
```
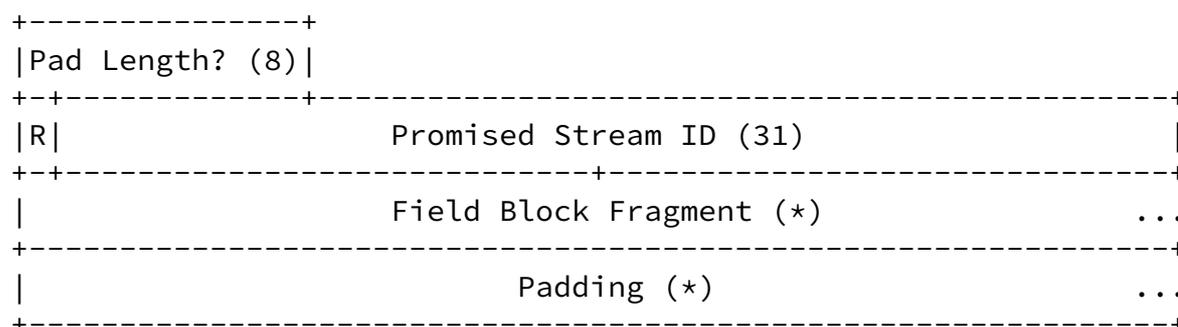
Figure 8: PUSH_PROMISE Frame Payload

The PUSH_PROMISE frame payload has the following fields:

Pad Length: An 8-bit field containing the length of the frame padding in units of octets. This field is only present if the PADDED flag is set.

R: A single reserved bit.

Promised Stream ID: An unsigned 31-bit integer that identifies the stream that is reserved by the PUSH_PROMISE. The promised stream identifier MUST be a valid choice for the next stream sent by the sender (see "new stream identifier" in Section 5.1.1).

Field Block Fragment: A field block fragment (Section 4.3) containing request control data and header section.

Padding: Padding octets.

The PUSH_PROMISE frame defines the following flags:

END_HEADERS (0x4): When set, bit 2 indicates that this frame
   contains an entire field block (Section 4.3) and is not followed
   by any CONTINUATION frames.

   A PUSH_PROMISE frame without the END_HEADERS flag set MUST be
   followed by a CONTINUATION frame for the same stream. A receiver
   MUST treat the receipt of any other type of frame or a frame on a
   different stream as a connection error (Section 5.4.1) of type
   PROTOCOL_ERROR.

PADDED (0x8): When set, bit 3 indicates that the Pad Length field
   and any padding that it describes are present.

PUSH_PROMISE frames MUST only be sent on a peer-initiated stream that
is in either the "open" or "half-closed (remote)" state. The stream
identifier of a PUSH_PROMISE frame indicates the stream it is
associated with. If the stream identifier field specifies the value
0x0, a recipient MUST respond with a connection error (Section 5.4.1)
of type PROTOCOL_ERROR.

Promised streams are not required to be used in the order they are
promised. The PUSH_PROMISE only reserves stream identifiers for
later use.

PUSH_PROMISE MUST NOT be sent if the SETTINGS_ENABLE_PUSH setting of
the peer endpoint is set to 0. An endpoint that has set this setting
and has received acknowledgement MUST treat the receipt of a
PUSH_PROMISE frame as a connection error (Section 5.4.1) of type
PROTOCOL_ERROR.

Recipients of PUSH_PROMISE frames can choose to reject promised
streams by returning a RST_STREAM referencing the promised stream
identifier back to the sender of the PUSH_PROMISE.

A PUSH_PROMISE frame modifies the connection state in two ways.

First, the inclusion of a field block ([Section 4.3](#)) potentially
modifies the state maintained for field section compression. Second,
PUSH_PROMISE also reserves a stream for later use, causing the
promised stream to enter the "reserved" state. A sender MUST NOT
send a PUSH_PROMISE on a stream unless that stream is either "open"
or "half-closed (remote)"; the sender MUST ensure that the promised
stream is a valid choice for a new stream identifier ([Section 5.1.1](#))
(that is, the promised stream MUST be in the "idle" state).

Since PUSH_PROMISE reserves a stream, ignoring a PUSH_PROMISE frame
causes the stream state to become indeterminate. A receiver MUST
treat the receipt of a PUSH_PROMISE on a stream that is neither
"open" nor "half-closed (local)" as a connection error
([Section 5.4.1](#)) of type PROTOCOL_ERROR. However, an endpoint that
has sent RST_STREAM on the associated stream MUST handle PUSH_PROMISE
frames that might have been created before the RST_STREAM frame is
received and processed.

A receiver MUST treat the receipt of a PUSH_PROMISE that promises an
illegal stream identifier ([Section 5.1.1](#)) as a connection error
([Section 5.4.1](#)) of type PROTOCOL_ERROR. Note that an illegal stream
identifier is an identifier for a stream that is not currently in the
"idle" state.

The PUSH_PROMISE frame can include padding. Padding fields and flags
are identical to those defined for DATA frames ([Section 6.1](#)).

6.7. PING

The PING frame (type=0x6) is a mechanism for measuring a minimal
round-trip time from the sender, as well as determining whether an
idle connection is still functional. PING frames can be sent from
any endpoint.

```
+---------------------------------------------------------------+
|                                                               |
|                      Opaque Data (64)                         |
|                                                               |
+---------------------------------------------------------------+
```

Figure 9: PING Frame Payload

In addition to the frame header, PING frames MUST contain 8 octets of
opaque data in the frame payload. A sender can include any value it
chooses and use those octets in any fashion.

Receivers of a PING frame that does not include an ACK flag MUST send a PING frame with the ACK flag set in response, with an identical frame payload. PING responses SHOULD be given higher priority than any other frame.

The PING frame defines the following flags:

ACK (0x1): When set, bit 0 indicates that this PING frame is a PING response. An endpoint MUST set this flag in PING responses. An endpoint MUST NOT respond to PING frames containing this flag.

PING frames are not associated with any individual stream. If a PING frame is received with a stream identifier field value other than 0x0, the recipient MUST respond with a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

Receipt of a PING frame with a length field value other than 8 MUST be treated as a connection error (Section 5.4.1) of type FRAME_SIZE_ERROR.

6.8. GOAWAY

The GOAWAY frame (type=0x7) is used to initiate shutdown of a connection or to signal serious error conditions. GOAWAY allows an endpoint to gracefully stop accepting new streams while still finishing processing of previously established streams. This enables administrative actions, like server maintenance.

There is an inherent race condition between an endpoint starting new streams and the remote sending a GOAWAY frame. To deal with this case, the GOAWAY contains the stream identifier of the last peer-initiated stream that was or might be processed on the sending endpoint in this connection. For instance, if the server sends a GOAWAY frame, the identified stream is the highest-numbered stream initiated by the client.

Once sent, the sender will ignore frames sent on streams initiated by the receiver if the stream has an identifier higher than the included last stream identifier. Receivers of a GOAWAY frame MUST NOT open additional streams on the connection, although a new connection can be established for new streams.

If the receiver of the GOAWAY has sent data on streams with a higher stream identifier than what is indicated in the GOAWAY frame, those streams are not or will not be processed. The receiver of the GOAWAY frame can treat the streams as though they had never been created at

all, thereby allowing those streams to be retried later on a new
connection.

Endpoints SHOULD always send a GOAWAY frame before closing a
connection 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 connection, the client
cannot know if the server started to process that POST request if the
server does not send a GOAWAY frame to indicate what streams it might
have acted on.

An endpoint might choose to close a connection without sending a
GOAWAY for misbehaving peers.

A GOAWAY frame might not immediately precede closing of the
connection; a receiver of a GOAWAY that has no more use for the
connection SHOULD still send a GOAWAY frame before terminating the
connection.

```
 +-+-------------------------------------------------------------+
 |R|                  Last-Stream-ID (31)                        |
 +-+-------------------------------------------------------------+
 |                      Error Code (32)                          |
 +---------------------------------------------------------------+
 |                  Additional Debug Data (*)                    |
 +---------------------------------------------------------------+
```
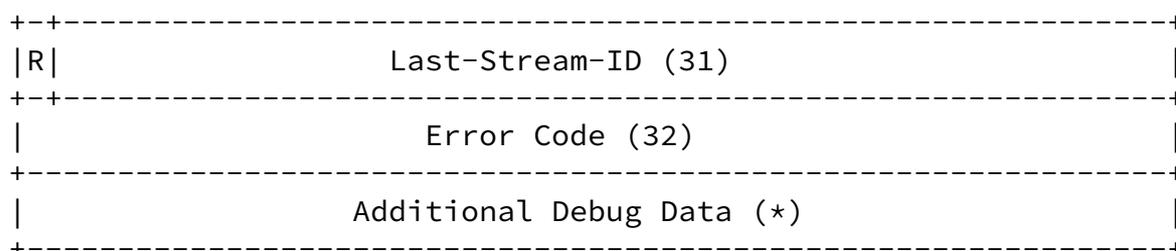
Figure 10: GOAWAY Frame Payload

The GOAWAY frame does not define any flags.

The GOAWAY frame applies to the connection, not a specific stream.
An endpoint MUST treat a GOAWAY frame with a stream identifier other
than 0x0 as a connection error (Section 5.4.1) of type
PROTOCOL_ERROR.

The last stream identifier in the GOAWAY frame contains the highest-
numbered stream identifier for which the sender of the GOAWAY frame
might have taken some action on or might yet take action on. All
streams up to and including the identified stream might have been
processed in some way. The last stream identifier can be set to 0 if
no streams were 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 connection terminates without a GOAWAY frame, the last stream
identifier is effectively the highest possible stream identifier.

On streams with lower- or equal-numbered identifiers that were not
closed completely prior to the connection being closed, reattempting
requests, transactions, or any protocol activity is not possible,
with the exception of idempotent actions like HTTP GET, PUT, or
DELETE. Any protocol activity that uses higher-numbered streams can
be safely retried using a new connection.

Activity on streams numbered lower or equal to the last stream
identifier might still complete successfully. The sender of a GOAWAY
frame might gracefully shut down a connection by sending a GOAWAY
frame, maintaining the connection in an "open" state until all in-
progress streams complete.

An endpoint MAY send multiple GOAWAY frames if circumstances change.
For instance, an endpoint that sends GOAWAY with NO_ERROR during
graceful shutdown could subsequently encounter a condition that
requires immediate termination of the connection. The last stream
identifier from the last GOAWAY frame received indicates which
streams could have been acted upon. Endpoints MUST NOT increase the
value they send in the last stream identifier, since the peers might
already have retried unprocessed requests on another connection.

A client that is unable to retry requests loses all requests that are
in flight when the server closes the connection. This is especially
true for intermediaries that might not be serving clients using
HTTP/2. A server that is attempting to gracefully shut down a
connection SHOULD send an initial GOAWAY frame with the last stream
identifier set to 2^31-1 and a NO_ERROR code. This signals to the
client that a shutdown is imminent and that initiating further
requests is prohibited. After allowing time for any in-flight stream
creation (at least one round-trip time), the server can send another
GOAWAY frame with an updated last stream identifier. This ensures

that a connection can be cleanly shut down without losing requests.

After sending a GOAWAY frame, the sender can discard frames for
streams initiated by the receiver with identifiers higher than the
identified last stream. However, any frames that alter connection
state cannot be completely ignored. For instance, HEADERS,
PUSH_PROMISE, and CONTINUATION frames MUST be minimally processed to
ensure the state maintained for field section compression is
consistent (see Section 4.3); similarly, DATA frames MUST be counted
toward the connection flow-control window. Failure to process these
frames can cause flow control or field section compression state to
become unsynchronized.

The GOAWAY frame also contains a 32-bit error code (Section 7) that
contains the reason for closing the connection.

Endpoints MAY append opaque data to the frame payload of any GOAWAY
frame. Additional debug data is intended for diagnostic purposes
only and carries no semantic value. Debug information could contain
security- or privacy-sensitive data. Logged or otherwise
persistently stored debug data MUST have adequate safeguards to
prevent unauthorized access.

6.9. WINDOW_UPDATE

The WINDOW_UPDATE frame (type=0x8) is used to implement flow control;
see Section 5.2 for an overview.

Flow control operates at two levels: on each individual stream and on
the entire connection.

Both types of flow control are hop by hop, that is, only between the
two endpoints. Intermediaries do not forward WINDOW_UPDATE frames
between dependent connections. However, throttling of data transfer
by any receiver can indirectly cause the propagation of flow-control
information toward the original sender.

Flow control only applies to frames that are identified as being
subject to flow control. Of the frame types defined in this
document, this includes only DATA frames. Frames that are exempt
from flow control MUST be accepted and processed, unless the receiver

is unable to assign resources to handling the frame. A receiver MAY
respond with a stream error (Section 5.4.2) or connection error
(Section 5.4.1) of type FLOW_CONTROL_ERROR if it is unable to accept
a frame.

```
+-+----------------------------------------------------------------+
|R|                 Window Size Increment (31)                     |
+-+----------------------------------------------------------------+
```
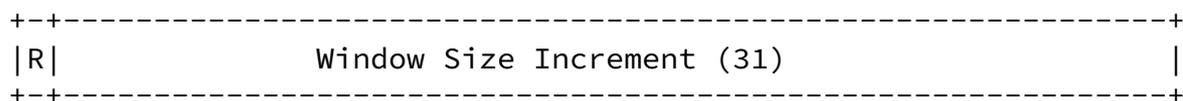
                    Figure 11: WINDOW_UPDATE Frame Payload

The frame payload of a WINDOW_UPDATE frame is one reserved bit plus
an unsigned 31-bit integer indicating the number of octets that the
sender can transmit in addition to the existing flow-control window.
The legal range for the increment to the flow-control window is 1 to
2^31-1 (2,147,483,647) octets.

The WINDOW_UPDATE frame does not define any flags.

The WINDOW_UPDATE frame can be specific to a stream or to the entire
connection. In the former case, the frame's stream identifier
indicates the affected stream; in the latter, the value "0" indicates
that the entire connection is the subject of the frame.

A receiver MUST treat the receipt of a WINDOW_UPDATE frame with an
flow-control window increment of 0 as a stream error (Section 5.4.2)
of type PROTOCOL_ERROR; errors on the connection flow-control window
MUST be treated as a connection error (Section 5.4.1).

WINDOW_UPDATE can be sent by a peer that has sent a frame bearing the
END_STREAM flag. This means that a receiver could receive a
WINDOW_UPDATE frame on a "half-closed (remote)" or "closed" stream.
A receiver MUST NOT treat this as an error (see Section 5.1).

A receiver that receives a flow-controlled frame MUST always account
for its contribution against the connection flow-control window,
unless the receiver treats this as a connection error
(Section 5.4.1). This is necessary even if the frame is in error.
The sender counts the frame toward the flow-control window, but if
the receiver does not, the flow-control window at the sender and
receiver can become different.

A WINDOW_UPDATE frame with a length other than 4 octets MUST be
treated as a connection error ([Section 5.4.1](#)) of type
FRAME_SIZE_ERROR.

6.9.1. The Flow-Control Window

Flow control in HTTP/2 is implemented using a window kept by each
sender on every stream. The flow-control window is a simple integer
value that indicates how many octets of data the sender is permitted
to transmit; as such, its size is a measure of the buffering capacity
of the receiver.

Two flow-control windows are applicable: the stream flow-control
window and the connection flow-control window. The sender MUST NOT
send a flow-controlled frame with a length that exceeds the space
available in either of the flow-control windows advertised by the
receiver. Frames with zero length with the END_STREAM flag set (that
is, an empty DATA frame) MAY be sent if there is no available space
in either flow-control window.

For flow-control calculations, the 9-octet frame header is not
counted.

After sending a flow-controlled frame, the sender reduces the space
available in both windows by the length of the transmitted frame.

The receiver of a frame sends a WINDOW_UPDATE frame as it consumes
data and frees up space in flow-control windows. Separate
WINDOW_UPDATE frames are sent for the stream- and connection-level
flow-control windows.

A sender that receives a WINDOW_UPDATE frame updates the
corresponding window by the amount specified in the frame.

A sender MUST NOT allow a flow-control window to exceed $2^{31}-1$
octets. If a sender receives a WINDOW_UPDATE that causes a flow-
control window to exceed this maximum, it MUST terminate either the
stream or the connection, as appropriate. For streams, the sender
sends a RST_STREAM with an error code of FLOW_CONTROL_ERROR; for the
connection, a GOAWAY frame with an error code of FLOW_CONTROL_ERROR
is sent.

Flow-controlled frames from the sender and WINDOW_UPDATE frames from
the receiver are completely asynchronous with respect to each other.
This property allows a receiver to aggressively update the window
size kept by the sender to prevent streams from stalling.

6.9.2. Initial Flow-Control Window Size

When an HTTP/2 connection is first established, new streams are
created with an initial flow-control window size of 65,535 octets.
The connection flow-control window is also 65,535 octets. Both
endpoints can adjust the initial window size for new streams by
including a value for SETTINGS_INITIAL_WINDOW_SIZE in the SETTINGS
frame that forms part of the connection preface. The connection
flow-control window can only be changed using WINDOW_UPDATE frames.

Prior to receiving a SETTINGS frame that sets a value for
SETTINGS_INITIAL_WINDOW_SIZE, an endpoint can only use the default
initial window size when sending flow-controlled frames. Similarly,
the connection flow-control window is set to the default initial
window size until a WINDOW_UPDATE frame is received.

In addition to changing the flow-control window for streams that are
not yet active, a SETTINGS frame can alter the initial flow-control
window size for streams with active flow-control windows (that is,
streams in the "open" or "half-closed (remote)" state). When the
value of SETTINGS_INITIAL_WINDOW_SIZE changes, a receiver MUST adjust
the size of all stream flow-control windows that it maintains by the
difference between the new value and the old value.

A change to SETTINGS_INITIAL_WINDOW_SIZE can cause the available
space in a flow-control window to become negative. A sender MUST
track the negative flow-control window and MUST NOT send new flow-
controlled frames until it receives WINDOW_UPDATE frames that cause
the flow-control window to become positive.

For example, if the client sends 60 KB immediately on connection
establishment and the server sets the initial window size to be 16
KB, the client will recalculate the available flow-control window to
be -44 KB on receipt of the SETTINGS frame. The client retains a

negative flow-control window until WINDOW_UPDATE frames restore the
window to being positive, after which the client can resume sending.

A SETTINGS frame cannot alter the connection flow-control window.

An endpoint MUST treat a change to SETTINGS_INITIAL_WINDOW_SIZE that
causes any flow-control window to exceed the maximum size as a
connection error (Section 5.4.1) of type FLOW_CONTROL_ERROR.

6.9.3. Reducing the Stream Window Size

A receiver that wishes to use a smaller flow-control window than the
current size can send a new SETTINGS frame. However, the receiver
MUST be prepared to receive data that exceeds this window size, since
the sender might send data that exceeds the lower limit prior to
processing the SETTINGS frame.

After sending a SETTINGS frame that reduces the initial flow-control
window size, a receiver MAY continue to process streams that exceed
flow-control limits. Allowing streams to continue does not allow the
receiver to immediately reduce the space it reserves for flow-control
windows. Progress on these streams can also stall, since
WINDOW_UPDATE frames are needed to allow the sender to resume
sending. The receiver MAY instead send a RST_STREAM with an error
code of FLOW_CONTROL_ERROR for the affected streams.

6.10. CONTINUATION

The CONTINUATION frame (type=0x9) is used to continue a sequence of
field block fragments (Section 4.3). Any number of CONTINUATION
frames can be sent, as long as the preceding frame is on the same
stream and is a HEADERS, PUSH_PROMISE, or CONTINUATION frame without
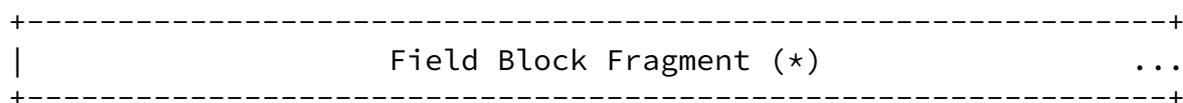the END_HEADERS flag set.

```
 +---------------------------------------------------------------+
 |                   Field Block Fragment (*)                  ...
 +---------------------------------------------------------------+
```

               Figure 12: CONTINUATION Frame Payload

The CONTINUATION frame payload contains a field block fragment
(Section 4.3).

The CONTINUATION frame defines the following flag:

END_HEADERS (0x4): When set, bit 2 indicates that this frame ends a
   field block (Section 4.3).

   If the END_HEADERS bit is not set, this frame MUST be followed by
   another CONTINUATION frame. A receiver MUST treat the receipt of
   any other type of frame or a frame on a different stream as a
   connection error (Section 5.4.1) of type PROTOCOL_ERROR.

The CONTINUATION frame changes the connection state as defined in
Section 4.3.

CONTINUATION frames MUST be associated with a stream. If a
CONTINUATION frame is received whose stream identifier field is 0x0,
the recipient MUST respond with a connection error (Section 5.4.1) of
type PROTOCOL_ERROR.

A CONTINUATION frame MUST be preceded by a HEADERS, PUSH_PROMISE or
CONTINUATION frame without the END_HEADERS flag set. A recipient
that observes violation of this rule MUST respond with a connection
error (Section 5.4.1) of type PROTOCOL_ERROR.

7. Error Codes

   Error codes are 32-bit fields that are used in RST_STREAM and GOAWAY
   frames to convey the reasons for the stream or connection error.

   Error codes share a common code space. Some error codes apply only
   to either streams or the entire connection and have no defined
   semantics in the other context.

   The following error codes are defined:

   NO_ERROR (0x0): The associated condition is not a result of an
      error. For example, a GOAWAY might include this code to indicate
      graceful shutdown of a connection.

   PROTOCOL_ERROR (0x1): The endpoint detected an unspecific protocol
      error. This error is for use when a more specific error code is
      not available.

   INTERNAL_ERROR (0x2): The endpoint encountered an unexpected
      internal error.

   FLOW_CONTROL_ERROR (0x3): The endpoint detected that its peer
      violated the flow-control protocol.

   SETTINGS_TIMEOUT (0x4): The endpoint sent a SETTINGS frame but did

not receive a response in a timely manner. See Section 6.5.3
("Settings Synchronization").

STREAM_CLOSED (0x5): The endpoint received a frame after a stream
   was half-closed.

FRAME_SIZE_ERROR (0x6): The endpoint received a frame with an
   invalid size.

REFUSED_STREAM (0x7): The endpoint refused the stream prior to
   performing any application processing (see Section 8.1.4 for
   details).

CANCEL (0x8): Used by the endpoint to indicate that the stream is no
   longer needed.

COMPRESSION_ERROR (0x9): The endpoint is unable to maintain the
   field section compression context for the connection.

CONNECT_ERROR (0xa): The connection established in response to a
   CONNECT request (Section 8.3) was reset or abnormally closed.

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

INADEQUATE_SECURITY (0xc): The underlying transport has properties
   that do not meet minimum security requirements (see Section 9.2).

HTTP_1_1_REQUIRED (0xd): The endpoint requires that HTTP/1.1 be used
   instead of HTTP/2.

Unknown or unsupported error codes MUST NOT trigger any special
behavior. These MAY be treated by an implementation as being
equivalent to INTERNAL_ERROR.

8. HTTP Message Exchanges

HTTP/2 defines a framing of the HTTP message abstraction (Section 6
of [HTTP]).

. HTTP Message Framing

   A client sends an HTTP request on a new stream, using a previously
   unused stream identifier (Section 5.1.1). A server sends an HTTP
   response on the same stream as the request.

   An HTTP message (request or response) consists of:

   1. one HEADERS frame (followed by zero or more CONTINUATION frames)
       containing the header section (see Section 6.3 of [HTTP]),

   2. zero or more DATA frames containing the message content (see
       Section 6.4 of [HTTP]), and

   3. optionally, one HEADERS frame, followed by zero or more
       CONTINUATION frames containing the trailer-part, if present (see
       Section 6.5 of [HTTP]).

   For a response only, a server MAY send any number of interim
   responses before the HEADERS frame containing a final response. An
   interim response consists of a HEADERS frames (which might be
   followed by zero or more CONTINUATION frames) containing the control
   data and header section of an interim (1xx) HTTP response (see
   Section 15 of [HTTP]). A HEADERS frame with an END_STREAM flag that
   carries an informational status code is malformed (Section 8.1.2.6).

   The last frame in the sequence bears an END_STREAM flag, noting that
   a HEADERS frame bearing the END_STREAM flag can be followed by
   CONTINUATION frames that carry any remaining fragments of the field
   block.

   Other frames (from any stream) MUST NOT occur between the HEADERS
   frame and any CONTINUATION frames that might follow.

   HTTP/2 uses DATA frames to carry message content. The "chunked"
   transfer encoding defined in Section 7.1 of [HTTP11] cannot be used
   in HTTP/2.

   Trailer fields are carried in a field block that also terminates the
   stream. That is, trailer fields comprise a sequence starting with a
   HEADERS frame, followed by zero or more CONTINUATION frames, where

the HEADERS frame bears an END_STREAM flag. Trailers MUST NOT
include pseudo-header fields ([Section 8.1.2.1](#)). An endpoint that
receives pseudo-header fields in trailers MUST treat the request or
response as malformed ([Section 8.1.2.6](#)).

An endpoint that receives a HEADERS frame without the END_STREAM flag
set after receiving the HEADERS frame that opens a request or after
receiving a final (non-informational) status code MUST treat the
corresponding request or response as malformed ([Section 8.1.2.6](#)).

An HTTP request/response exchange fully consumes a single stream. A
request starts with the HEADERS frame that puts the stream into an
"open" state. The request ends with a frame bearing END_STREAM,
which causes the stream to become "half-closed (local)" for the
client and "half-closed (remote)" for the server. A response stream
starts with zero or more interim responses in HEADERS frames or a
HEADERS frame containing a final status code.

An HTTP response is complete after the server sends -- or the client
receives -- a frame with the END_STREAM flag set (including any
CONTINUATION frames needed to complete a field block). 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 sending a RST_STREAM with an error code of NO_ERROR after sending
a complete response (i.e., a frame with the END_STREAM flag).
Clients MUST NOT discard responses as a result of receiving such a
RST_STREAM, though clients can always discard responses at their
discretion for other reasons.

8.1.1. Upgrading from HTTP/2

HTTP/2 removes support for the 101 (Switching Protocols)
informational status code (Section 15.2.2 of [HTTP]).

The semantics of 101 (Switching Protocols) aren't applicable to a multiplexed protocol. Alternative protocols are able to use the same mechanisms that HTTP/2 uses to negotiate their use (see Section 3).

8.1.2. HTTP Fields

HTTP fields carry information as a series of field lines, which are key-value pairs. For a listing of registered HTTP fields, see the "Hypertext Transfer Protocol (HTTP) Field Name Registry" registry maintained at https://www.iana.org/assignments/http-fields/.

Field names are strings of ASCII characters that are compared in a case-insensitive fashion. Field names MUST be converted to lowercase when constructing a HTTP/2 message. A request or response containing an uppercase character ('A' to 'Z', ASCII 0x41 to 0x5a) in a field name MUST be treated as malformed (Section 8.1.2.6).

HPACK is capable of carrying field names or values that are not valid in HTTP. Though HPACK can carry any octet, fields are not valid in the following cases:

* A field name MUST NOT contain characters in the range 0x00-0x20 or 0x7F-0xFF (both ranges inclusive). This limits field names to visible ASCII characters, other than ASCII SP (0x20).

* With the exception of pseudo-header fields (Section 8.1.2.1), which have a name that starts with a single colon, field names MUST NOT include a colon (ASCII COLON, 0x3a).

* A field value MUST NOT contain the zero value (ASCII NUL, 0x0), line feed (ASCII LF, 0xa), or carriage return (ASCII CR, 0xd) at any position.

* A field value MUST NOT start or end with an ASCII whitespace character (ASCII SP or HTAB, 0x20 or 0x9).

A request or response that contains a field that violates any of these conditions MUST be treated as malformed (Section 8.1.2.6). In particular, an intermediary that does not process fields when forwarding messages MUST NOT forward fields that contain any of the

values that are listed as prohibited above.

Field values that are not valid according to the definition of the
corresponding field do not cause a request to be malformed except as
defined by the processing rules for the field.

8.1.2.1. Pseudo-Header Fields

HTTP/2 uses special pseudo-header fields beginning with ':' character
(ASCII 0x3a) to convey message control data (see Section 6.2 of
[HTTP]).

Pseudo-header fields are not HTTP header fields. Endpoints MUST NOT
generate pseudo-header fields other than those defined in this
document. Note that an extension could negotiate the use of
additional pseudo-header fields; see Section 5.5.

Pseudo-header fields are only valid in the context in which they are
defined. Pseudo-header fields defined for requests MUST NOT appear
in responses; pseudo-header fields defined for responses MUST NOT
appear in requests. Pseudo-header fields MUST NOT appear in a
trailer section. Endpoints MUST treat a request or response that
contains undefined or invalid pseudo-header fields as malformed
(Section 8.1.2.6).

All pseudo-header fields MUST appear in a field block before all
regular field lines. Any request or response that contains a pseudo-
header field that appears in a field block after a regular field line
MUST be treated as malformed (Section 8.1.2.6).

8.1.2.2. Connection-Specific Header Fields

HTTP/2 does not use the "Connection" header field to indicate
connection-specific header fields; in this protocol, connection-
specific metadata is conveyed by other means. An endpoint MUST NOT
generate an HTTP/2 message containing connection-specific header
fields; any message containing connection-specific header fields MUST
be treated as malformed (Section 8.1.2.6).

The only exception to this is the TE header field, which MAY be
present in an HTTP/2 request; when it is, it MUST NOT contain any
value other than "trailers".

An intermediary transforming a HTTP/1.x message to HTTP/2 MUST remove connection-specific header fields as discussed in Section 7.6.1 of [HTTP], or their messages will be treated by other HTTP/2 endpoints as malformed (Section 8.1.2.6).

> | Note: HTTP/2 purposefully does not support upgrade to another
> | protocol. The handshake methods described in Section 3 are
> | believed sufficient to negotiate the use of alternative
> | protocols.

8.1.2.3. Request Pseudo-Header Fields

The following pseudo-header fields are defined for HTTP/2 requests:

*   The ":method" pseudo-header field includes the HTTP method (Section 9 of [HTTP]).

*   The ":scheme" pseudo-header field includes the scheme portion of the request target. The scheme is taken from the target URI (Section 3.1 of [RFC3986]) when generating a request directly, or from the scheme of a translated request (for example. see Section 3.3 of [HTTP11]). Scheme is omitted for CONNECT requests (Section 8.3).

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

*   The ":authority" pseudo-header field includes the authority portion of the target URI (Section 3.2 of [RFC3986]). The authority MUST NOT include the deprecated "userinfo" subcomponent for "http" or "https" schemed URIs.

Clients that generate HTTP/2 requests directly SHOULD use the ":authority" pseudo-header field instead of the "Host" header field.

An intermediary that translates a request to HTTP/2 from another

HTTP version MUST translate any authority information from the
request into an ":authority" pseudo-header field. If the control
data in the original request contains authority information, an
intermediary MUST include a ":authority" pseudo-header field. If
control data does not contain authority, an intermediary MUST NOT
add an ":authority" pseudo-header field. For reference, an
HTTP/1.1 Section 3.2 of request target [HTTP11] in authority-form
always includes authority, a request target in absolute-form
includes authority if the target URI includes authority, and
request targets in origin- or asterisk-form do not include
authority.

An intermediary that translates a request to another HTTP version
from HTTP/2 can construct a "Host" header field by copying the
value of the ":authority" pseudo-header field if that version
requires that "Host" be included in a request, as HTTP/1.1 does
for some forms of request target (see Section 3.2 of [HTTP11]).

An intermediary that translates a request to HTTP/2 from another
HTTP version MUST retain any "Host" header field, even if an
authority is part of control data.

The value of the "Host" header field MUST be ignored if control
data contains authority (that is, the ":authority" pseudo-header
field is present).

* The ":path" pseudo-header field includes the path and query parts
   of the target URI (the "path-absolute" production and optionally a
   '?' character followed by the "query" production (see Sections 3.3
   and 3.4 of [RFC3986]). A request in asterisk form includes the
   value '*' for the ":path" pseudo-header field.

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

All HTTP/2 requests MUST include exactly one valid value for the
":method", ":scheme", and ":path" pseudo-header fields, unless it is
a CONNECT request (Section 8.3). An HTTP request that omits
mandatory pseudo-header fields is malformed (Section 8.1.2.6).

Thomson & Benfield   Expires 4 December 2021   [Page 51]

Individual HTTP/2 requests do not carry an explicit indicator of
protocol version. All HTTP/2 messages implicitly have a protocol
version of "2.0" (see Section 6.2 of [HTTP]).

8.1.2.4. Response Pseudo-Header Fields

For HTTP/2 responses, a single ":status" pseudo-header field is
defined that carries the HTTP status code field (see Section 15 of
[HTTP]). This pseudo-header field MUST be included in all responses,
including interim responses; otherwise, the response is malformed
(Section 8.1.2.6).

HTTP/2 responses implicitly have a protocol version of "2.0".

8.1.2.5. Compressing the Cookie Header Field

The Cookie header field [COOKIE] uses a semi-colon (";") to delimit
cookie-pairs (or "crumbs"). This header field contains multiple
values, but does not use a COMMA (",") as a separator, which prevents
cookie-pairs from being sent on multiple field lines (see Section 5.2
of [HTTP]). This can significantly reduce compression efficiency as
updates to individual cookie-pairs would invalidate any field lines
that are stored in the HPACK table.

To allow for better compression efficiency, the Cookie header field
MAY be split into separate header fields, each with one or more
cookie-pairs. If there are multiple Cookie header fields after
decompression, these MUST be concatenated into a single octet string
using the two-octet delimiter of 0x3B, 0x20 (the ASCII string "; ")
before being passed into a non-HTTP/2 context, such as an HTTP/1.1
connection, or a generic HTTP server application.

Therefore, the following two lists of Cookie header fields are
semantically equivalent.

```
cookie: a=b; c=d; e=f

cookie: a=b
cookie: c=d
cookie: e=f
```

8.1.2.6. Malformed Requests and Responses

A malformed request or response is one that is an otherwise valid
sequence of HTTP/2 frames but is invalid due to the presence of
extraneous frames, prohibited fields or pseudo-header fields, the
absence of mandatory fields or pseudo-header fields, or the inclusion
of uppercase field names.

A request or response that includes message content 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
content. A response that is defined to have no content, as described
in Section 6.4 of [HTTP], 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 5.4.2) of type
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.

Endpoints that progressively process messages might have performed
some processing before identifying a request or response as
malformed. For instance, it might be possible to generate an
informational or 404 status code without having received a complete
request. Similarly, intermediaries might forward incomplete messages
before detecting errors. A server MAY generate a final response
before receiving an entire request when the response does not depend
on the remainder of the request being correct. A server or
intermediary MAY use RST_STREAM -- with a code other than
REFUSED_STREAM -- to abort a stream if a malformed request or
response is received.

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.

8.1.3. Examples

This section shows HTTP/1.1 requests and responses, with
illustrations of equivalent HTTP/2 requests and responses.

An HTTP GET request includes control data and a request header with
no message content and is therefore transmitted as a single HEADERS
frame, followed by zero or more CONTINUATION frames containing the

serialized block of request header fields. The HEADERS frame in the
following has both the END_HEADERS and END_STREAM flags set; no
CONTINUATION frames are sent.

```
    GET /resource HTTP/1.1           HEADERS
    Host: example.org        ==>       + END_STREAM
    Accept: image/jpeg                 + END_HEADERS
                                          :method = GET
                                          :scheme = https
                                          :path = /resource
                                          host = example.org
                                          accept = image/jpeg
```

Similarly, a response that includes only control data and a response
header is transmitted as a HEADERS frame (again, followed by zero or
more CONTINUATION frames) containing the serialized block of response
header fields.

```
    HTTP/1.1 304 Not Modified         HEADERS
    ETag: "xyzzy"            ==>        + END_STREAM
    Expires: Thu, 23 Jan ...           + END_HEADERS
                                          :status = 304
                                          etag = "xyzzy"
                                          expires = Thu, 23 Jan ...
```

An HTTP POST request that includes control data and a request header
and message content is transmitted as one HEADERS frame, followed by
zero or more CONTINUATION frames containing the request header,
followed by one or more DATA frames, with the last CONTINUATION (or
HEADERS) frame having the END_HEADERS flag set and the final DATA
frame having the END_STREAM flag set:

```
    POST /resource HTTP/1.1           HEADERS
    Host: example.org        ==>       - END_STREAM
    Content-Type: image/jpeg           - END_HEADERS
    Content-Length: 123                   :method = POST
                                          :path = /resource
    {binary data}                         :scheme = https
```

```
                                   CONTINUATION
                                     + END_HEADERS
                                       content-type = image/jpeg
                                       host = example.org
                                       content-length = 123

                                   DATA
                                     + END_STREAM
                                   {binary data}
```

   Note that data contributing to any given field line could be spread
   between field block fragments. The allocation of field lines to
   frames in this example is illustrative only.

   A response that includes control data and a response header and
   message content is transmitted as a HEADERS frame, followed by zero
   or more CONTINUATION frames, followed by one or more DATA frames,
   with the last DATA frame in the sequence having the END_STREAM flag
   set:

```
   HTTP/1.1 200 OK              HEADERS
   Content-Type: image/jpeg   ==>    - END_STREAM
   Content-Length: 123               + END_HEADERS
                                       :status = 200
   {binary data}                       content-type = image/jpeg
                                       content-length = 123

                               DATA
                                 + END_STREAM
                               {binary data}
```

   An informational response using a 1xx status code other than 101 is
   transmitted as a HEADERS frame, followed by zero or more CONTINUATION
   frames.

   A trailer section is sent as a field block after both the request or
   response field block and all the DATA frames have been sent. The
   HEADERS frame starting the field block that comprises the trailer
   section has the END_STREAM flag set.

   The following example includes both a 100 (Continue) status code,
   which is sent in response to a request containing a "100-continue"

token in the Expect header field, and a trailer section:

```
HTTP/1.1 100 Continue            HEADERS
Extension-Field: bar      ==>      - END_STREAM
                                   + END_HEADERS
                                     :status = 100
                                     extension-field = bar


HTTP/1.1 200 OK                  HEADERS
Content-Type: image/jpeg   ==>     - END_STREAM
Transfer-Encoding: chunked         + END_HEADERS
Trailer: Foo                         :status = 200
                                     content-length = 123
123                                  content-type = image/jpeg
{binary data}                        trailer = Foo
0
Foo: bar                         DATA
                                   - END_STREAM
                                 {binary data}

                                 HEADERS
                                   + END_STREAM
                                   + END_HEADERS
```

foo = bar

8.1.4. Request Reliability Mechanisms in HTTP/2

   In general, an HTTP client is unable to retry a non-idempotent
   request when an error occurs because there is no means to determine
   the nature of the error. It is possible that some server processing
   occurred prior to the error, which could result in undesirable
   effects if the request were reattempted.

   HTTP/2 provides two mechanisms for providing a guarantee to a client
   that a request has not been processed:

   * The GOAWAY frame indicates the highest stream number that might
      have been processed. Requests on streams with higher numbers are
      therefore guaranteed to be safe to retry.

   * The REFUSED_STREAM error code can be included in a RST_STREAM
      frame to indicate that the stream is being closed prior to any
      processing having occurred. Any request that was sent on the
      reset stream can be safely retried.

   Requests that have not been processed have not failed; clients MAY
   automatically retry them, even those with non-idempotent methods.

   A server MUST NOT indicate that a stream has not been processed
   unless it can guarantee that fact. If frames that are on a stream
   are passed to the application layer for any stream, then
   REFUSED_STREAM MUST NOT be used for that stream, and a GOAWAY frame
   MUST include a stream identifier that is greater than or equal to the
   given stream identifier.

   In addition to these mechanisms, the PING frame provides a way for a
   client to easily test a connection. Connections that remain idle can
   become broken as some middleboxes (for instance, network address
   translators or load balancers) silently discard connection bindings.
   The PING frame allows a client to safely test whether a connection is
   still active without sending a request.

8.2. Server Push

   HTTP/2 allows a server to pre-emptively send (or "push") responses
   (along with corresponding "promised" requests) to a client in
   association with a previous client-initiated request.

   Server push was designed to allow a server to improve client-
   perceived performance by predicting what requests will follow those
   that it receives, thereby removing a round trip for them. For
   example, a request for HTML is often followed by requests for
   stylesheets and scripts referenced by that page. When these requests
   are pushed, the client does not need to wait to receive the
   references to them in the HTML and issue separate requests.

   In practice, server push is difficult to use effectively, because it
   requires the server to correctly anticipate the additional requests
   the client will make, taking into account factors such as caching,
   content negotiation, and user behavior. Errors in prediction can
   lead to performance degradation, due to the opportunity cost that the
   additional data on the wire represents. In particular, pushing any
   significant amount of data can cause contention issues with more-
   important responses.

   A client can request that server push be disabled, though this is
   negotiated for each hop independently. The SETTINGS_ENABLE_PUSH
   setting can be set to 0 to indicate that server push is disabled.

   Promised requests MUST be safe (see Section 9.2.1 of [HTTP]) and
   cacheable (see Section 9.2.3 of [HTTP]). Promised requests cannot
   include any content or a trailer section. Clients that receive a
   promised request that is not cacheable, that is not known to be safe,
   or that indicates the presence of request content MUST reset the
   promised stream with a stream error (Section 5.4.2) of type
   PROTOCOL_ERROR. Note this could result in the promised stream being

reset if the client does not recognize a newly defined method as
being safe.

Pushed responses that are cacheable (see Section 3 of [CACHE]) can be
stored by the client, if it implements an HTTP cache. Pushed
responses are considered successfully validated on the origin server
(e.g., if the "no-cache" cache response directive is present; see
Section 5.2.2.3 of [CACHE]) while the stream identified by the
promised stream ID is still open.

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

The server MUST include a value in the ":authority" pseudo-header
field for which the server is authoritative (see Section 10.1). A
client MUST treat a PUSH_PROMISE for which the server is not
authoritative as a stream error (Section 5.4.2) of type
PROTOCOL_ERROR.

An intermediary can receive pushes from the server and choose not to
forward them on to the client. In other words, how to make use of
the pushed information is up to that intermediary. Equally, the
intermediary might choose to make additional pushes to the client,
without any action taken by the server.

A client cannot push. Thus, servers MUST treat the receipt of a
PUSH_PROMISE frame as a connection error (Section 5.4.1) of type
PROTOCOL_ERROR. A server cannot set the SETTINGS_ENABLE_PUSH setting
to a value other than 0 (see Section 6.5.2).

8.2.1. Push Requests

Server push is semantically equivalent to a server responding to a
request; however, in this case, that request is also sent by the
server, as a PUSH_PROMISE frame.

The PUSH_PROMISE frame includes a field block that contains control
data and a complete set of request header fields that the server
attributes to the request. It is not possible to push a response to
a request that includes message content.

Promised requests are always associated with an explicit request from the client. The PUSH_PROMISE frames sent by the server are sent on that explicit request's stream. The PUSH_PROMISE frame also includes a promised stream identifier, chosen from the stream identifiers available to the server (see Section 5.1.1).

The header fields in PUSH_PROMISE and any subsequent CONTINUATION frames MUST be a valid and complete set of request header fields (Section 8.1.2.3). The server MUST include a method in the ":method" pseudo-header field that is safe and cacheable. If a client receives a PUSH_PROMISE that does not include a complete and valid set of header fields or the ":method" pseudo-header field identifies a method that is not safe, it MUST respond with a stream error (Section 5.4.2) of type PROTOCOL_ERROR.

The server SHOULD send PUSH_PROMISE (Section 6.6) frames prior to sending any frames that reference the promised responses. This avoids a race where clients issue requests prior to receiving any PUSH_PROMISE frames.

For example, if the server receives a request for a document containing embedded links to multiple image files and the server chooses to push those additional images to the client, sending PUSH_PROMISE frames before the DATA frames that contain the image links ensures that the client is able to see that a resource will be pushed before discovering embedded links. Similarly, if the server pushes responses referenced by the field block (for instance, in Link header fields), sending a PUSH_PROMISE before sending the header ensures that clients do not request those resources.

PUSH_PROMISE frames MUST NOT be sent by the client.

PUSH_PROMISE frames can be sent by the server in response to any client-initiated stream, but the stream MUST be in either the "open" or "half-closed (remote)" state with respect to the server. PUSH_PROMISE frames are interspersed with the frames that comprise a response, though they cannot be interspersed with HEADERS and CONTINUATION frames that comprise a single field block.

Sending a PUSH_PROMISE frame creates a new stream and puts the stream into the "reserved (local)" state for the server and the "reserved (remote)" state for the client.

8.2.2. Push Responses

   After sending the PUSH_PROMISE frame, the server can begin delivering
   the pushed response as a response (Section 8.1.2.4) on a server-
   initiated stream that uses the promised stream identifier. The
   server uses this stream to transmit an HTTP response, using the same
   sequence of frames as defined in Section 8.1. This stream becomes
   "half-closed" to the client (Section 5.1) after the initial HEADERS
   frame is sent.

   Once a client receives a PUSH_PROMISE frame and chooses to accept the
   pushed response, the client SHOULD NOT issue any requests for the
   promised response until after the promised stream has closed.

   If the client determines, for any reason, that it does not wish to
   receive the pushed response from the server or if the server takes
   too long to begin sending the promised response, the client can send
   a RST_STREAM frame, using either the CANCEL or REFUSED_STREAM code
   and referencing the pushed stream's identifier.

   A client can use the SETTINGS_MAX_CONCURRENT_STREAMS setting to limit
   the number of responses that can be concurrently pushed by a server.
   Advertising a SETTINGS_MAX_CONCURRENT_STREAMS value of zero disables
   server push by preventing the server from creating the necessary
   streams. This does not prohibit a server from sending PUSH_PROMISE
   frames; clients need to reset any promised streams that are not
   wanted.

   Clients receiving a pushed response MUST validate that either the
   server is authoritative (see Section 10.1) or the proxy that provided
   the pushed response is configured for the corresponding request. For
   example, a server that offers a certificate for only the
   "example.com" DNS-ID is not permitted to push a response for
   "https://www.example.org/doc".

   The response for a PUSH_PROMISE stream begins with a HEADERS frame,
   which immediately puts the stream into the "half-closed (remote)"
   state for the server and "half-closed (local)" state for the client,
   and ends with a frame bearing END_STREAM, which places the stream in
   the "closed" state.

   | Note: The client never sends a frame with the END_STREAM flag
   | for a server push.

8.3. The CONNECT Method

   In HTTP/1.x, the pseudo-method CONNECT (Section 9.3.6 of [HTTP]) is
   used to convert an HTTP connection into a tunnel to a remote host.
   CONNECT 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/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 header section is constructed as defined in Section 8.1.2.3
   ("Request Pseudo-Header Fields"), with a few differences.
   Specifically:

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

   * The ":scheme" and ":path" pseudo-header fields MUST be omitted.

   * The ":authority" pseudo-header field contains the host and port to
      connect to (equivalent to the authority-form of the request-target
      of CONNECT requests; see Section 3.2.3 of [HTTP11]).

   A CONNECT request that does not conform to these restrictions is
   malformed (Section 8.1.2.6).

   A proxy that supports CONNECT establishes a TCP connection [TCP] to
   the host and port identified in the ":authority" pseudo-header field.
   Once this connection is successfully established, the proxy sends a
   HEADERS frame containing a 2xx series status code to the client, as
   defined in Section 9.3.6 of [HTTP].

   After the initial HEADERS frame sent by each peer, all subsequent
   DATA frames correspond to data sent on the TCP connection. The frame
   payload of any DATA frames sent by the client is transmitted by the
   proxy to the TCP server; data received from the TCP server is
   assembled into DATA frames by the proxy. Frame types other than DATA

or stream management frames (RST_STREAM, WINDOW_UPDATE, and PRIORITY)
MUST NOT be sent on a connected stream and MUST be treated as a
stream error (Section 5.4.2) if received.

The TCP connection can be closed by either peer. The END_STREAM flag
on a DATA frame is treated as being equivalent to the TCP FIN bit. A
client is expected to send a DATA frame with the END_STREAM flag set
after receiving a frame bearing the END_STREAM flag. A proxy that
receives a DATA frame with the END_STREAM flag set sends the attached
data with the FIN bit set on the last TCP segment. A proxy that
receives a TCP segment with the FIN bit set sends a DATA frame with
the END_STREAM flag set. Note that the final TCP segment or DATA
frame could be empty.

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 (Section 5.4.2) of
type CONNECT_ERROR. Correspondingly, a proxy MUST send a TCP segment
with the RST bit set if it detects an error with the stream or the
HTTP/2 connection.

9. Additional HTTP Requirements/Considerations

This section outlines attributes of the HTTP protocol that improve
interoperability, reduce exposure to known security vulnerabilities,
or reduce the potential for implementation variation.

9.1. Connection Management

HTTP/2 connections are persistent. For best performance, it is
expected that clients will not close connections until it is
determined that no further communication with a server is necessary
(for example, when a user navigates away from a particular web page)

or until the server closes the connection.

Clients SHOULD NOT open more than one HTTP/2 connection to a given
host and port pair, where the host is derived from a URI, a selected
alternative service [ALT-SVC], or a configured proxy.

A client can create additional connections as replacements, either to
replace connections that are near to exhausting the available stream
identifier space (Section 5.1.1), to refresh the keying material for
a TLS connection, or to replace connections that have encountered
errors (Section 5.4.1).

A client MAY open multiple connections to the same IP address and TCP
port using different Server Name Indication [TLS-EXT] values or to
provide different TLS client certificates but SHOULD avoid creating
multiple connections with the same configuration.

Servers are encouraged to maintain open connections for as long as
possible but are permitted to terminate idle connections if
necessary. When either endpoint chooses to close the transport-layer
TCP connection, the terminating endpoint SHOULD first send a GOAWAY
(Section 6.8) frame so that both endpoints can reliably determine
whether previously sent frames have been processed and gracefully
complete or terminate any necessary remaining tasks.

9.1.1. Connection Reuse

Connections that are made to an origin server, either directly or
through a tunnel created using the CONNECT method (Section 8.3), MAY
be reused for requests with multiple different URI authority
components. A connection can be reused as long as the origin server
is authoritative (Section 10.1). For TCP connections without TLS,
this depends on the host having resolved to the same IP address.

For "https" resources, connection reuse additionally depends on
having a certificate that is valid for the host in the URI. The
certificate presented by the server MUST satisfy any checks that the
client would perform when forming a new TLS connection for the host
in the URI. A single certificate can be used to establish authority

for multiple origins. Section 4.3 of [HTTP] describes how a client
determines whether a server is authoritative for a URI.

In some deployments, reusing a connection for multiple origins can
result in requests being directed to the wrong origin server. For
example, TLS termination might be performed by a middlebox that uses
the TLS Server Name Indication (SNI) [TLS-EXT] extension to select an
origin server. This means that it is possible for clients to send
requests to servers that might not be the intended target for the
request, even though the server is otherwise authoritative.

A server that does not wish clients to reuse connections 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 15.5.20 of [HTTP]).

A client that is configured to use a proxy over HTTP/2 directs
requests to that proxy through a single connection. That is, all
requests sent via a proxy reuse the connection to the proxy.

9.2. Use of TLS Features

Implementations of HTTP/2 MUST use TLS version 1.2 [TLS12] or higher
for HTTP/2 over TLS. The general TLS usage guidance in [TLSBCP]
SHOULD be followed, with some additional restrictions that are
specific to HTTP/2.

The TLS implementation MUST support the Server Name Indication (SNI)
[TLS-EXT] extension to TLS. If the server is identified by a domain
name [DNS-TERMS], clients MUST send the server_name TLS extension
unless an alternative mechanism to indicate the target host is used.

Requirements for deployments of HTTP/2 that negotiate TLS 1.3 [TLS13]
are included in Section 9.2.3. Deployments of TLS 1.2 are subject to
the requirements in Section 9.2.1 and Section 9.2.2. Implementations
are encouraged to provide defaults that comply, but it is recognized
that deployments are ultimately responsible for compliance.

9.2.1. TLS 1.2 Features

This section describes restrictions on the TLS 1.2 feature set that
can be used with HTTP/2. Due to deployment limitations, it might not

be possible to fail TLS negotiation when these restrictions are not
met. An endpoint MAY immediately terminate an HTTP/2 connection that
does not meet these TLS requirements with a connection error
(Section 5.4.1) of type INADEQUATE_SECURITY.

A deployment of HTTP/2 over TLS 1.2 MUST disable compression. TLS
compression can lead to the exposure of information that would not
otherwise be revealed [RFC3749]. Generic compression is unnecessary
since HTTP/2 provides compression features that are more aware of
context and therefore likely to be more appropriate for use for
performance, security, or other reasons.

A deployment of HTTP/2 over TLS 1.2 MUST disable renegotiation. An
endpoint MUST treat a TLS renegotiation as a connection error
(Section 5.4.1) of type PROTOCOL_ERROR. Note that disabling
renegotiation can result in long-lived connections becoming unusable
due to limits on the number of messages the underlying cipher suite
can encipher.

An endpoint MAY use renegotiation to provide confidentiality
protection for client credentials offered in the handshake, but any
renegotiation MUST occur prior to sending the connection preface. A
server SHOULD request a client certificate if it sees a renegotiation
request immediately after establishing a connection.

This effectively prevents the use of renegotiation in response to a
request for a specific protected resource. A future specification
might provide a way to support this use case. Alternatively, a
server might use an error (Section 5.4) of type HTTP_1_1_REQUIRED to
request the client use a protocol that supports renegotiation.

Implementations MUST support ephemeral key exchange sizes of at least
2048 bits for cipher suites that use ephemeral finite field Diffie-
Hellman (DHE) [TLS13] and 224 bits for cipher suites that use
ephemeral elliptic curve Diffie-Hellman (ECDHE) [RFC4492]. Clients
MUST accept DHE sizes of up to 4096 bits. Endpoints MAY treat
negotiation of key sizes smaller than the lower limits as a
connection error (Section 5.4.1) of type INADEQUATE_SECURITY.

. TLS 1.2 Cipher Suites

   A deployment of HTTP/2 over TLS 1.2 SHOULD NOT use any of the cipher
   suites that are listed in the list of prohibited cipher suites
   (Appendix A).

   Endpoints MAY choose to generate a connection error ([Section 5.4.1](#))
   of type INADEQUATE_SECURITY if one of the prohibited cipher suites is
   negotiated. A deployment that chooses to use a prohibited cipher
   suite risks triggering a connection error unless the set of potential
   peers is known to accept that cipher suite.

   Implementations MUST NOT generate this error in reaction to the
   negotiation of a cipher suite that is not prohibited. Consequently,
   when clients offer a cipher suite that is not prohibited, they have
   to be prepared to use that cipher suite with HTTP/2.

   The list of prohibited cipher suites includes the cipher suite that
   TLS 1.2 makes mandatory, which means that TLS 1.2 deployments could
   have non-intersecting sets of permitted cipher suites. To avoid this
   problem causing TLS handshake failures, deployments of HTTP/2 that
   use TLS 1.2 MUST support TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
   [TLS-ECDHE] with the P-256 elliptic curve [FIPS186].

   Note that clients might advertise support of cipher suites that are
   prohibited in order to allow for connection to servers that do not
   support HTTP/2. This allows servers to select HTTP/1.1 with a cipher
   suite that is prohibited in HTTP/2. However, this can result in
   HTTP/2 being negotiated with a prohibited cipher suite if the
   application protocol and cipher suite are independently selected.

9.2.3. TLS 1.3 Features

   TLS 1.3 includes a number of features not available in earlier
   versions. This section discusses the use of these features.

   HTTP/2 servers MUST NOT send post-handshake TLS 1.3
   CertificateRequest messages. HTTP/2 clients MUST treat a TLS post-
   handshake CertificateRequest message as a connection error
   ([Section 5.4.1](#)) of type PROTOCOL_ERROR.

   The prohibition on post-handshake authentication applies even if the

client offered the "post_handshake_auth" TLS extension. Post-
handshake authentication support might be advertised independently of
ALPN [TLS-ALPN]. Clients might offer the capability for use in other
protocols, but inclusion of the extension cannot imply support within
HTTP/2.

[TLS13] defines other post-handshake messages, NewSessionTicket and
KeyUpdate, which can be used as they have no direct interaction with
HTTP/2. Unless the use of a new type of TLS message depends on an
interaction with the application-layer protocol, that TLS message can
be sent after the handshake completes.

TLS early data MAY be used to send requests, provided that the
guidance in [RFC8470] is observed. Clients send requests in early
data assuming initial values for all server settings.

10. Security Considerations

10.1. Server Authority

HTTP/2 relies on the HTTP definition of authority for determining
whether a server is authoritative in providing a given response (see
Section 4.3 of [HTTP]). This relies on local name resolution for the
"http" URI scheme and the authenticated server identity for the
"https" scheme.

10.2. Cross-Protocol Attacks

In a cross-protocol attack, an attacker causes a client to initiate a
transaction in one protocol toward a server that understands a
different protocol. An attacker might be able to cause the
transaction to appear as a valid transaction in the second protocol.
In combination with the capabilities of the web context, this can be
used to interact with poorly protected servers in private networks.

Completing a TLS handshake with an ALPN identifier for HTTP/2 can be
considered sufficient protection against cross-protocol attacks.
ALPN provides a positive indication that a server is willing to
proceed with HTTP/2, which prevents attacks on other TLS-based
protocols.

The encryption in TLS makes it difficult for attackers to control the
data that could be used in a cross-protocol attack on a cleartext
protocol.

The cleartext version of HTTP/2 has minimal protection against cross-
protocol attacks. The connection preface (Section 3.4) contains a
string that is designed to confuse HTTP/1.1 servers, but no special
protection is offered for other protocols.

10.3. Intermediary Encapsulation Attacks

HPACK permits encoding of field names and values that might be
treated as delimiters in other HTTP versions. An intermediary that
translates an HTTP/2 request or response MUST validate fields
according to the rules in Section 8.1.2 roles before translating a
message to another HTTP version. Translating a field that includes
invalid delimiters could be used to cause recipients to incorrectly
interpret a message, which could be exploited by an attacker.

An intermediary can reject fields that contain invalid field names or
values for other reasons, in particular those that do not conform to
the HTTP ABNF grammar from Section 5 of [HTTP]. Intermediaries that
do not perform any validation of fields other than the minimum
required by Section 8.1.2 could forward messages that contain invalid
field names or values.

An intermediary that receives any field that requires removal before
forwarding (see Section 7.6.1 of [HTTP]) MUST remove or replace those
header fields when forwarding messages. Additionally, intermediaries
should take care when forwarding messages containing Content-Length
fields to ensure that the message is well-formed (Section 8.1.2.6).
This ensures that if the message is translated into HTTP/1.1 at any
point the framing will be correct.

10.4. Cacheability of Pushed Responses

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

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

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

authoritative tenant provides.

   Pushed responses for which an origin server is not authoritative (see
   Section 10.1) MUST NOT be used or cached.

10.5. Denial-of-Service Considerations

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

   The number of PUSH_PROMISE frames is not constrained in the same
   fashion. A client that accepts server push SHOULD limit the number
   of streams it allows to be in the "reserved (remote)" state. An
   excessive number of server push streams can be treated as a stream
   error (Section 5.4.2) of type ENHANCE_YOUR_CALM.

   A number of HTTP/2 implementations were found to be vulnerable to
   denial of service [NFLX-2019-002]. The following lists known ways
   that implementations might be subject to denial of service attack:

   * Inefficient tracking of outstanding outbound frames can lead to
      overload if an adversary can cause large numbers of frames to be
      enqueued for sending. A peer could use one of several techniques
      to cause large numbers of frames to be generated:

      - Providing tiny increments to flow control in WINDOW_UPDATE
         frames can cause a sender to generate a large number of DATA
         frames.

      - An endpoint is required to respond to a PING frame.

      - Each SETTINGS frame requires acknowledgment.

      - An invalid request (or server push) can cause a peer to send
         RST_STREAM frames in response.

   * Large numbers of small or empty frames can be abused to cause a

peer to expend time processing frame headers. Caution is required
here as some uses of small frames are entirely legitimate, such as
the sending of an empty DATA or CONTINUATION frame at the end of a
stream.

* The SETTINGS frame might also be abused to cause a peer to expend
   additional processing time. This might be done by pointlessly
   changing settings, sending multiple undefined settings, or
   changing the same setting multiple times in the same frame.

* Handling reprioritization with PRIORITY frames can require
   significant processing time and can lead to overload if many
   PRIORITY frames are sent.

* Field section compression also offers some opportunities to waste
   processing resources; see Section 7 of [COMPRESSION] for more
   details on potential abuses.

* Limits in SETTINGS cannot be reduced instantaneously, which leaves
   an endpoint exposed to behavior from a peer that could exceed the
   new limits. In particular, immediately after establishing a
   connection, limits set by a server are not known to clients and
   could be exceeded without being an obvious protocol violation.

* An attacker can provide large amounts of flow control credit at
   the HTTP/2 layer, but withhold credit at the TCP layer, preventing
   frames from being sent. An endpoint that constructs and remembers
   frames for sending without considering TCP limits might be subject
   to resource exhaustion.

Most of the features that might be exploited for denial of service --
i.e., SETTINGS changes, small frames, field section compression --
have legitimate uses. These features become a burden only when they
are used unnecessarily or to excess.

An endpoint that doesn't monitor use of these features exposes itself
to a risk of denial of service. Implementations SHOULD track the use
of these features and set limits on their use. An endpoint MAY treat
activity that is suspicious as a connection error (Section 5.4.1) of
type ENHANCE_YOUR_CALM.

10.5.1. Limits on Field Block Size

A large field block (Section 4.3) can cause an implementation to
commit a large amount of state. Field lines that are critical for
routing can appear toward the end of a field block, which prevents
streaming of fields to their ultimate destination. This ordering and
other reasons, such as ensuring cache correctness, mean that an
endpoint might need to buffer the entire field block. Since there is
no hard limit to the size of a field block, some endpoints could be
forced to commit a large amount of available memory for field blocks.

An endpoint can use the SETTINGS_MAX_HEADER_LIST_SIZE to advise peers
of limits that might apply on the size of uncompressed field blocks.
This setting is only advisory, so endpoints MAY choose to send field
blocks that exceed this limit and risk having the request or response
being treated as malformed. This setting is specific to a
connection, so any request or response could encounter a hop with a

lower, unknown limit. An intermediary can attempt to avoid this
problem by passing on values presented by different peers, but they
are not obliged to do so.

A server that receives a larger field block 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 field block MUST be processed to ensure a consistent
connection state, unless the connection is closed.

10.5.2. CONNECT Issues

The CONNECT method can be used to create disproportionate load on an
proxy, since stream creation is relatively inexpensive when compared
to the creation and maintenance of a TCP connection. A proxy might
also maintain some resources for a TCP connection beyond the closing
of the stream that carries the CONNECT request, since the outgoing
TCP connection remains in the TIME_WAIT state. Therefore, a proxy
cannot rely on SETTINGS_MAX_CONCURRENT_STREAMS alone to limit the
resources consumed by CONNECT requests.

10.6. Use of Compression

Compression can allow an attacker to recover secret data when it is
compressed in the same context as data under attacker control.

HTTP/2 enables compression of field lines (Section 4.3); the
following concerns also apply to the use of HTTP compressed content-
codings (Section 8.4.1 of [HTTP]).

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

Implementations communicating on a secure channel MUST NOT compress
content that includes both confidential and attacker-controlled data
unless separate compression dictionaries are used for each source of
data. Compression MUST NOT be used if the source of data cannot be
reliably determined. Generic stream compression, such as that
provided by TLS, MUST NOT be used with HTTP/2 (see Section 9.2).

Further considerations regarding the compression of header fields are
described in [COMPRESSION].

10.7. Use of Padding

Padding within HTTP/2 is not intended as a replacement for general
purpose padding, such as that provided by TLS [TLS13]. Redundant
padding could even be counterproductive. Correct application can
depend on having specific knowledge of the data that is being padded.

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

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

Use of padding can result in less protection than might seem
immediately obvious. At best, padding only makes it more difficult
for an attacker to infer length information by increasing the number

of frames an attacker has to observe. Incorrectly implemented
padding schemes can be easily defeated. In particular, randomized
padding with a predictable distribution provides very little
protection; similarly, padding frame payloads to a fixed size exposes
information as frame payload sizes cross the fixed-sized boundary,
which could be possible if an attacker can control plaintext.

Intermediaries SHOULD retain padding for DATA frames but MAY drop
padding for HEADERS and PUSH_PROMISE frames. A valid reason for an
intermediary to change the amount of padding of frames is to improve
the protections that padding provides.

10.8. Privacy Considerations

Several characteristics of HTTP/2 provide an observer an opportunity
to correlate actions of a single client or server over time. These
include the value of settings, the manner in which flow-control
windows are managed, the way priorities are allocated to streams, the
timing of reactions to stimulus, and the handling of any features
that are controlled by settings.

As far as these create observable differences in behavior, they could
be used as a basis for fingerprinting a specific client, as defined
in Section 3.2 of [PRIVACY].

HTTP/2's preference for using a single TCP connection allows
correlation of a user's activity on a site. Reusing connections for
different origins allows tracking across those origins.

Because the PING and SETTINGS frames solicit immediate responses,
they can be used by an endpoint to measure latency to their peer.
This might have privacy implications in certain scenarios.

10.9. Remote Timing Attacks

Remote timing attacks extract secrets from servers by observing
variations in the time that servers take when processing requests
that use secrets. HTTP/2 enables concurrent request creation and
processing, which can give attackers better control over when request
processing commences. Multiple HTTP/2 requests can be included in

the same IP packet or TLS record. HTTP/2 can therefore make remote
timing attacks more efficient by eliminating variability in request
delivery, leaving only request order and the delivery of responses as
sources of timing variability.

Ensuring that processing time is not dependent on the value of
secrets is the best defense against any form of timing attack.

11. IANA Considerations

A string for identifying HTTP/2 is entered into the "Application-
Layer Protocol Negotiation (ALPN) Protocol IDs" registry established
in [TLS-ALPN].

This document establishes a registry for frame types, settings, and
error codes. These new registries appear in the new "Hypertext
Transfer Protocol version 2 (HTTP/2)" section.

This document registers the "HTTP2-Settings" header field for use in
HTTP.

This document registers the "PRI" method for use in HTTP to avoid
collisions with the connection preface (Section 3.4).

11.1. Registration of HTTP/2 Identification Strings

This document creates two registrations for the identification of
HTTP/2 (see Section 3.2) in the "Application-Layer Protocol
Negotiation (ALPN) Protocol IDs" registry established in [TLS-ALPN].

The "h2" string identifies HTTP/2 when used over TLS:

Protocol: HTTP/2 over TLS

Identification Sequence: 0x68 0x32 ("h2")

Specification: This document

Thomson & Benfield   Expires 4 December 2021   [Page 72]

The "h2c" string identifies HTTP/2 when used over cleartext TCP:

Protocol: HTTP/2 over TCP

Identification Sequence: 0x68 0x32 0x63 ("h2c")

   Specification: This document

11.2. Frame Type Registry

   This document establishes a registry for HTTP/2 frame type codes.
   The "HTTP/2 Frame Type" registry manages an 8-bit space. The "HTTP/2
   Frame Type" registry operates under either of the Section 4.8 of
   "IETF Review" [RFC8126] or Section 4.10 of "IESG Approval" [RFC8126]
   policies.

   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, its semantics, and flags that the
      frame type uses, including any parts of the frame that are
      conditionally present based on the value of flags.

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

```
+================+======+==============+
| Frame Type     | Code | Section      |
+================+======+==============+
| DATA           | 0x0  | Section 6.1  |
+----------------+------+--------------+
| HEADERS        | 0x1  | Section 6.2  |
+----------------+------+--------------+
| PRIORITY       | 0x2  | Section 6.3  |
+----------------+------+--------------+
| RST_STREAM     | 0x3  | Section 6.4  |
+----------------+------+--------------+
| SETTINGS       | 0x4  | Section 6.5  |
+----------------+------+--------------+
| PUSH_PROMISE   | 0x5  | Section 6.6  |
+----------------+------+--------------+
| PING           | 0x6  | Section 6.7  |
+----------------+------+--------------+
| GOAWAY         | 0x7  | Section 6.8  |
+----------------+------+--------------+
| WINDOW_UPDATE  | 0x8  | Section 6.9  |
+----------------+------+--------------+
| CONTINUATION   | 0x9  | Section 6.10 |
+----------------+------+--------------+
```

                     Table 1

11.3. Settings Registry

   This document establishes a registry for HTTP/2 settings. The
   "HTTP/2 Settings" registry manages a 16-bit space. The "HTTP/2
   Settings" registry operates under the Section 4.5 of "Expert Review"
   policy [RFC8126].

   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.

   Initial Value: An initial value for 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.

Name	Code	Initial Value	Specification
========================	======	===============	===============
HEADER_TABLE_SIZE	0x1	4096	Section 6.5.2
ENABLE_PUSH	0x2	1	Section 6.5.2
MAX_CONCURRENT_STREAMS	0x3	(infinite)	Section 6.5.2
INITIAL_WINDOW_SIZE	0x4	65535	Section 6.5.2
MAX_FRAME_SIZE	0x5	16384	Section 6.5.2
MAX_HEADER_LIST_SIZE	0x6	(infinite)	Section 6.5.2

Table 2

11.4. Error Code Registry

   This document establishes a registry for HTTP/2 error codes. The
   "HTTP/2 Error Code" registry manages a 32-bit space. The "HTTP/2
   Error Code" registry operates under the Section 4.5 of "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 32-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
======================	======	=======================	===============
NO_ERROR	0x0	Graceful shutdown	[Section 7](#)
PROTOCOL_ERROR	0x1	Protocol error detected	[Section 7](#)
INTERNAL_ERROR	0x2	Implementation fault	[Section 7](#)
FLOW_CONTROL_ERROR	0x3	Flow-control limits exceeded	[Section 7](#)
SETTINGS_TIMEOUT	0x4	Settings not acknowledged	[Section 7](#)
STREAM_CLOSED	0x5	Frame received for closed stream	[Section 7](#)
FRAME_SIZE_ERROR	0x6	Frame size incorrect	[Section 7](#)
REFUSED_STREAM	0x7	Stream not processed	[Section 7](#)
CANCEL	0x8	Stream cancelled	[Section 7](#)
COMPRESSION_ERROR	0x9	Compression state not updated	[Section 7](#)
CONNECT_ERROR	0xa	TCP connection error for CONNECT	[Section 7](#)

```
|                     |      | method               |              |
+---------------------+------+----------------------+--------------+
| ENHANCE_YOUR_CALM   | 0xb  | Processing           | Section 7    |
|                     |      | capacity exceeded    |              |
+---------------------+------+----------------------+--------------+
| INADEQUATE_SECURITY | 0xc  | Negotiated TLS       | Section 7    |
|                     |      | parameters not       |              |
|                     |      | acceptable           |              |
+---------------------+------+----------------------+--------------+
| HTTP_1_1_REQUIRED   | 0xd  | Use HTTP/1.1 for     | Section 7    |
|                     |      | the request          |              |
+---------------------+------+----------------------+--------------+
```

Table 3

11.5. HTTP2-Settings Header Field Registration

   This section marks the "HTTP2-Settings" header field registered in
   Section 11.5 of [RFC7540] as obsoleted.

11.6. PRI Method Registration

   This section registers the "PRI" method in the "HTTP Method Registry"
   (Section 18.2 of [HTTP]).

   Method Name: PRI

   Safe: Yes

   Idempotent: Yes

   Specification document(s): Section 3.4 of this document

   Related information: This method is never used by an actual client.
      This method will appear to be used when an HTTP/1.1 server or
      intermediary attempts to parse an HTTP/2 connection preface.

11.7. The h2c Upgrade Token

   Previous versions of this document (Section 11.8 of [RFC7540])
   registered an upgrade token. This capability has been removed: see

Section 3.1.

12. References

12.1. Normative References

    [CACHE]   Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke,
                Ed., "HTTP Caching", Work in Progress, Internet-Draft,
                draft-ietf-httpbis-cache-15, 30 March 2021,
                <https://tools.ietf.org/html/draft-ietf-httpbis-cache-15>.

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

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

    [FIPS186]   NIST, "Digital Signature Standard (DSS)", FIPS PUB 186-4,
                July 2013, <http://dx.doi.org/10.6028/NIST.FIPS.186-4>.

    [HTTP]   Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke,
                Ed., "HTTP Semantics", Work in Progress, Internet-Draft,
                draft-ietf-httpbis-semantics-15, 30 March 2021,
                <https://tools.ietf.org/html/draft-ietf-httpbis-semantics-
                15>.

    [RFC2119]   Bradner, S., "Key words for use in RFCs to Indicate
                Requirement Levels", BCP 14, RFC 2119, March 1997,
                <https://www.rfc-editor.org/rfc/rfc2119>.

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

    [RFC8126]   Cotton, M., Leiba, B., and R. 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/rfc/rfc8126>.

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

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

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

   [TLS-ECDHE]
              Rescorla, E., "TLS Elliptic Curve Cipher Suites with SHA-
              256/384 and AES Galois Counter Mode (GCM)", RFC 5289,
              August 2008, <https://www.rfc-editor.org/rfc/rfc5289>.

   [TLS-EXT] Eastlake 3rd, D., "Transport Layer Security (TLS)
              Extensions: Extension Definitions", RFC 6066, January
              2011, <https://www.rfc-editor.org/rfc/rfc6066>.

   [TLS12]   Dierks, T. and E. Rescorla, "The Transport Layer Security
              (TLS) Protocol Version 1.2", RFC 5246, August 2008,
              <https://www.rfc-editor.org/rfc/rfc5246>.

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

12.2. Informative References

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

   [BREACH]   Gluck, Y., Harris, N., and A. Prado, "BREACH: Reviving the
              CRIME Attack", 12 July 2013,

                 <http://breachattack.com/resources/
                 BREACH%20-%20SSL,%20gone%20in%2030%20seconds.pdf>.

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

     [HTTP11]   Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke,
                 Ed., "HTTP/1.1", Work in Progress, Internet-Draft, draft-
                 ietf-httpbis-messaging-15, 30 March 2021,
                 <https://tools.ietf.org/html/draft-ietf-httpbis-messaging-
                 15>.

     [I-D.ietf-httpbis-priority]
                 Oku, K. and L. Pardue, "Extensible Prioritization Scheme
                 for HTTP", Work in Progress, Internet-Draft, draft-ietf-
                 httpbis-priority-03, 11 January 2021,
                 <https://tools.ietf.org/html/draft-ietf-httpbis-priority-
                 03>.

     [NFLX-2019-002]
                 Netflix, "HTTP/2 Denial of Service Advisory", 13 August
                 2019, <https://github.com/Netflix/security-
                 bulletins/blob/master/advisories/third-party/2019-002.md>.

     [PRIVACY]   Cooper, A., Tschofenig, H., Aboba, B., Peterson, J.,
                 Morris, J., Hansen, M., and R. Smith, "Privacy
                 Considerations for Internet Protocols", RFC 6973,
                 DOI 10.17487/RFC6973, July 2013,
                 <https://www.rfc-editor.org/rfc/rfc6973>.

     [RFC3749]   Hollenbeck, S., "Transport Layer Security Protocol
                 Compression Methods", RFC 3749, May 2004,
                 <https://www.rfc-editor.org/rfc/rfc3749>.

     [RFC4492]   Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., and B.
                 Moeller, "Elliptic Curve Cryptography (ECC) Cipher Suites
                 for Transport Layer Security (TLS)", RFC 4492, May 2006,
                 <https://www.rfc-editor.org/rfc/rfc4492>.

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

   [RFC7323] Borman, D., Braden, B., Jacobson, V., and R.
              Scheffenegger, Ed., "TCP Extensions for High Performance",
              RFC 7323, September 2014,
              <https://www.rfc-editor.org/rfc/rfc7323>.

   [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/rfc/rfc7540>.

   [RFC8740] Benjamin, D., "Using TLS 1.3 with HTTP/2", RFC 8740,
              DOI 10.17487/RFC8740, February 2020,
              <https://www.rfc-editor.org/rfc/rfc8740>.

   [TALKING] Huang, L., Chen, E., Barth, A., Rescorla, E., and C.
              Jackson, "Talking to Yourself for Fun and Profit", 2011,
              <http://w2spconf.com/2011/papers/websocket.pdf>.

   [TLSBCP]   Sheffer, Y., Holz, R., and P. Saint-Andre,
              "Recommendations for Secure Use of Transport Layer
              Security (TLS) and Datagram Transport Layer Security
              (DTLS)", BCP 195, RFC 7525, May 2015,
              <https://www.rfc-editor.org/rfc/rfc7525>.

Appendix A. Prohibited TLS 1.2 Cipher Suites

   An HTTP/2 implementation MAY treat the negotiation of any of the
   following cipher suites with TLS 1.2 as a connection error
   (Section 5.4.1) of type INADEQUATE_SECURITY:

   * TLS_NULL_WITH_NULL_NULL
   * TLS_RSA_WITH_NULL_MD5
   * TLS_RSA_WITH_NULL_SHA
   * TLS_RSA_EXPORT_WITH_RC4_40_MD5
   * TLS_RSA_WITH_RC4_128_MD5
   * TLS_RSA_WITH_RC4_128_SHA
   * TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
   * TLS_RSA_WITH_IDEA_CBC_SHA
   * TLS_RSA_EXPORT_WITH_DES40_CBC_SHA

* TLS_RSA_WITH_DES_CBC_SHA
* TLS_RSA_WITH_3DES_EDE_CBC_SHA
* TLS_DH_DSS_EXPORT_WITH_DES40_CBC_SHA
* TLS_DH_DSS_WITH_DES_CBC_SHA
* TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA
* TLS_DH_RSA_EXPORT_WITH_DES40_CBC_SHA
* TLS_DH_RSA_WITH_DES_CBC_SHA
* TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA
* TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
* TLS_DHE_DSS_WITH_DES_CBC_SHA
* TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA
* TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
* TLS_DHE_RSA_WITH_DES_CBC_SHA
* TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA
* TLS_DH_anon_EXPORT_WITH_RC4_40_MD5
* TLS_DH_anon_WITH_RC4_128_MD5
* TLS_DH_anon_EXPORT_WITH_DES40_CBC_SHA
* TLS_DH_anon_WITH_DES_CBC_SHA
* TLS_DH_anon_WITH_3DES_EDE_CBC_SHA
* TLS_KRB5_WITH_DES_CBC_SHA
* TLS_KRB5_WITH_3DES_EDE_CBC_SHA
* TLS_KRB5_WITH_RC4_128_SHA
* TLS_KRB5_WITH_IDEA_CBC_SHA
* TLS_KRB5_WITH_DES_CBC_MD5
* TLS_KRB5_WITH_3DES_EDE_CBC_MD5
* TLS_KRB5_WITH_RC4_128_MD5
* TLS_KRB5_WITH_IDEA_CBC_MD5
* TLS_KRB5_EXPORT_WITH_DES_CBC_40_SHA
* TLS_KRB5_EXPORT_WITH_RC2_CBC_40_SHA
* TLS_KRB5_EXPORT_WITH_RC4_40_SHA
* TLS_KRB5_EXPORT_WITH_DES_CBC_40_MD5
* TLS_KRB5_EXPORT_WITH_RC2_CBC_40_MD5
* TLS_KRB5_EXPORT_WITH_RC4_40_MD5
* TLS_PSK_WITH_NULL_SHA
* TLS_DHE_PSK_WITH_NULL_SHA
* TLS_RSA_PSK_WITH_NULL_SHA
* TLS_RSA_WITH_AES_128_CBC_SHA
* TLS_DH_DSS_WITH_AES_128_CBC_SHA
* TLS_DH_RSA_WITH_AES_128_CBC_SHA
* TLS_DHE_DSS_WITH_AES_128_CBC_SHA
* TLS_DHE_RSA_WITH_AES_128_CBC_SHA
* TLS_DH_anon_WITH_AES_128_CBC_SHA
* TLS_RSA_WITH_AES_256_CBC_SHA
* TLS_DH_DSS_WITH_AES_256_CBC_SHA
* TLS_DH_RSA_WITH_AES_256_CBC_SHA
* TLS_DHE_DSS_WITH_AES_256_CBC_SHA
* TLS_DHE_RSA_WITH_AES_256_CBC_SHA
* TLS_DH_anon_WITH_AES_256_CBC_SHA

   *   TLS_RSA_WITH_NULL_SHA256
   *   TLS_RSA_WITH_AES_128_CBC_SHA256
   *   TLS_RSA_WITH_AES_256_CBC_SHA256
   *   TLS_DH_DSS_WITH_AES_128_CBC_SHA256
   *   TLS_DH_RSA_WITH_AES_128_CBC_SHA256
   *   TLS_DHE_DSS_WITH_AES_128_CBC_SHA256
   *   TLS_RSA_WITH_CAMELLIA_128_CBC_SHA
   *   TLS_DH_DSS_WITH_CAMELLIA_128_CBC_SHA
   *   TLS_DH_RSA_WITH_CAMELLIA_128_CBC_SHA
   *   TLS_DHE_DSS_WITH_CAMELLIA_128_CBC_SHA
   *   TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA
   *   TLS_DH_anon_WITH_CAMELLIA_128_CBC_SHA
   *   TLS_DHE_RSA_WITH_AES_128_CBC_SHA256
   *   TLS_DH_DSS_WITH_AES_256_CBC_SHA256
   *   TLS_DH_RSA_WITH_AES_256_CBC_SHA256
   *   TLS_DHE_DSS_WITH_AES_256_CBC_SHA256
   *   TLS_DHE_RSA_WITH_AES_256_CBC_SHA256
   *   TLS_DH_anon_WITH_AES_128_CBC_SHA256
   *   TLS_DH_anon_WITH_AES_256_CBC_SHA256
   *   TLS_RSA_WITH_CAMELLIA_256_CBC_SHA
   *   TLS_DH_DSS_WITH_CAMELLIA_256_CBC_SHA
   *   TLS_DH_RSA_WITH_CAMELLIA_256_CBC_SHA
   *   TLS_DHE_DSS_WITH_CAMELLIA_256_CBC_SHA
   *   TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA
   *   TLS_DH_anon_WITH_CAMELLIA_256_CBC_SHA
   *   TLS_PSK_WITH_RC4_128_SHA
   *   TLS_PSK_WITH_3DES_EDE_CBC_SHA
   *   TLS_PSK_WITH_AES_128_CBC_SHA
   *   TLS_PSK_WITH_AES_256_CBC_SHA
   *   TLS_DHE_PSK_WITH_RC4_128_SHA
   *   TLS_DHE_PSK_WITH_3DES_EDE_CBC_SHA
   *   TLS_DHE_PSK_WITH_AES_128_CBC_SHA
   *   TLS_DHE_PSK_WITH_AES_256_CBC_SHA
   *   TLS_RSA_PSK_WITH_RC4_128_SHA
   *   TLS_RSA_PSK_WITH_3DES_EDE_CBC_SHA
   *   TLS_RSA_PSK_WITH_AES_128_CBC_SHA
   *   TLS_RSA_PSK_WITH_AES_256_CBC_SHA
   *   TLS_RSA_WITH_SEED_CBC_SHA
   *   TLS_DH_DSS_WITH_SEED_CBC_SHA
   *   TLS_DH_RSA_WITH_SEED_CBC_SHA
   *   TLS_DHE_DSS_WITH_SEED_CBC_SHA
   *   TLS_DHE_RSA_WITH_SEED_CBC_SHA

*   TLS_DH_anon_WITH_SEED_CBC_SHA
   *   TLS_RSA_WITH_AES_128_GCM_SHA256
   *   TLS_RSA_WITH_AES_256_GCM_SHA384
   *   TLS_DH_RSA_WITH_AES_128_GCM_SHA256
   *   TLS_DH_RSA_WITH_AES_256_GCM_SHA384
   *   TLS_DH_DSS_WITH_AES_128_GCM_SHA256

   *   TLS_DH_DSS_WITH_AES_256_GCM_SHA384
   *   TLS_DH_anon_WITH_AES_128_GCM_SHA256
   *   TLS_DH_anon_WITH_AES_256_GCM_SHA384
   *   TLS_PSK_WITH_AES_128_GCM_SHA256
   *   TLS_PSK_WITH_AES_256_GCM_SHA384
   *   TLS_RSA_PSK_WITH_AES_128_GCM_SHA256
   *   TLS_RSA_PSK_WITH_AES_256_GCM_SHA384
   *   TLS_PSK_WITH_AES_128_CBC_SHA256
   *   TLS_PSK_WITH_AES_256_CBC_SHA384
   *   TLS_PSK_WITH_NULL_SHA256
   *   TLS_PSK_WITH_NULL_SHA384
   *   TLS_DHE_PSK_WITH_AES_128_CBC_SHA256
   *   TLS_DHE_PSK_WITH_AES_256_CBC_SHA384
   *   TLS_DHE_PSK_WITH_NULL_SHA256
   *   TLS_DHE_PSK_WITH_NULL_SHA384
   *   TLS_RSA_PSK_WITH_AES_128_CBC_SHA256
   *   TLS_RSA_PSK_WITH_AES_256_CBC_SHA384
   *   TLS_RSA_PSK_WITH_NULL_SHA256
   *   TLS_RSA_PSK_WITH_NULL_SHA384
   *   TLS_RSA_WITH_CAMELLIA_128_CBC_SHA256
   *   TLS_DH_DSS_WITH_CAMELLIA_128_CBC_SHA256
   *   TLS_DH_RSA_WITH_CAMELLIA_128_CBC_SHA256
   *   TLS_DHE_DSS_WITH_CAMELLIA_128_CBC_SHA256
   *   TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA256
   *   TLS_DH_anon_WITH_CAMELLIA_128_CBC_SHA256
   *   TLS_RSA_WITH_CAMELLIA_256_CBC_SHA256
   *   TLS_DH_DSS_WITH_CAMELLIA_256_CBC_SHA256
   *   TLS_DH_RSA_WITH_CAMELLIA_256_CBC_SHA256
   *   TLS_DHE_DSS_WITH_CAMELLIA_256_CBC_SHA256
   *   TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA256
   *   TLS_DH_anon_WITH_CAMELLIA_256_CBC_SHA256
   *   TLS_EMPTY_RENEGOTIATION_INFO_SCSV
   *   TLS_ECDH_ECDSA_WITH_NULL_SHA
   *   TLS_ECDH_ECDSA_WITH_RC4_128_SHA
   *   TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA

*   TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA
   *   TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA
   *   TLS_ECDHE_ECDSA_WITH_NULL_SHA
   *   TLS_ECDHE_ECDSA_WITH_RC4_128_SHA
   *   TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA
   *   TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
   *   TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
   *   TLS_ECDH_RSA_WITH_NULL_SHA
   *   TLS_ECDH_RSA_WITH_RC4_128_SHA
   *   TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA
   *   TLS_ECDH_RSA_WITH_AES_128_CBC_SHA
   *   TLS_ECDH_RSA_WITH_AES_256_CBC_SHA
   *   TLS_ECDHE_RSA_WITH_NULL_SHA

   *   TLS_ECDHE_RSA_WITH_RC4_128_SHA
   *   TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA
   *   TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
   *   TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
   *   TLS_ECDH_anon_WITH_NULL_SHA
   *   TLS_ECDH_anon_WITH_RC4_128_SHA
   *   TLS_ECDH_anon_WITH_3DES_EDE_CBC_SHA
   *   TLS_ECDH_anon_WITH_AES_128_CBC_SHA
   *   TLS_ECDH_anon_WITH_AES_256_CBC_SHA
   *   TLS_SRP_SHA_WITH_3DES_EDE_CBC_SHA
   *   TLS_SRP_SHA_RSA_WITH_3DES_EDE_CBC_SHA
   *   TLS_SRP_SHA_DSS_WITH_3DES_EDE_CBC_SHA
   *   TLS_SRP_SHA_WITH_AES_128_CBC_SHA
   *   TLS_SRP_SHA_RSA_WITH_AES_128_CBC_SHA
   *   TLS_SRP_SHA_DSS_WITH_AES_128_CBC_SHA
   *   TLS_SRP_SHA_WITH_AES_256_CBC_SHA
   *   TLS_SRP_SHA_RSA_WITH_AES_256_CBC_SHA
   *   TLS_SRP_SHA_DSS_WITH_AES_256_CBC_SHA
   *   TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
   *   TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384
   *   TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256
   *   TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA384
   *   TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
   *   TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
   *   TLS_ECDH_RSA_WITH_AES_128_CBC_SHA256
   *   TLS_ECDH_RSA_WITH_AES_256_CBC_SHA384
   *   TLS_ECDH_ECDSA_WITH_AES_128_GCM_SHA256
   *   TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384

* TLS_ECDH_RSA_WITH_AES_128_GCM_SHA256
   * TLS_ECDH_RSA_WITH_AES_256_GCM_SHA384
   * TLS_ECDHE_PSK_WITH_RC4_128_SHA
   * TLS_ECDHE_PSK_WITH_3DES_EDE_CBC_SHA
   * TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA
   * TLS_ECDHE_PSK_WITH_AES_256_CBC_SHA
   * TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
   * TLS_ECDHE_PSK_WITH_AES_256_CBC_SHA384
   * TLS_ECDHE_PSK_WITH_NULL_SHA
   * TLS_ECDHE_PSK_WITH_NULL_SHA256
   * TLS_ECDHE_PSK_WITH_NULL_SHA384
   * TLS_RSA_WITH_ARIA_128_CBC_SHA256
   * TLS_RSA_WITH_ARIA_256_CBC_SHA384
   * TLS_DH_DSS_WITH_ARIA_128_CBC_SHA256
   * TLS_DH_DSS_WITH_ARIA_256_CBC_SHA384
   * TLS_DH_RSA_WITH_ARIA_128_CBC_SHA256
   * TLS_DH_RSA_WITH_ARIA_256_CBC_SHA384
   * TLS_DHE_DSS_WITH_ARIA_128_CBC_SHA256
   * TLS_DHE_DSS_WITH_ARIA_256_CBC_SHA384
   * TLS_DHE_RSA_WITH_ARIA_128_CBC_SHA256

   * TLS_DHE_RSA_WITH_ARIA_256_CBC_SHA384
   * TLS_DH_anon_WITH_ARIA_128_CBC_SHA256
   * TLS_DH_anon_WITH_ARIA_256_CBC_SHA384
   * TLS_ECDHE_ECDSA_WITH_ARIA_128_CBC_SHA256
   * TLS_ECDHE_ECDSA_WITH_ARIA_256_CBC_SHA384
   * TLS_ECDH_ECDSA_WITH_ARIA_128_CBC_SHA256
   * TLS_ECDH_ECDSA_WITH_ARIA_256_CBC_SHA384
   * TLS_ECDHE_RSA_WITH_ARIA_128_CBC_SHA256
   * TLS_ECDHE_RSA_WITH_ARIA_256_CBC_SHA384
   * TLS_ECDH_RSA_WITH_ARIA_128_CBC_SHA256
   * TLS_ECDH_RSA_WITH_ARIA_256_CBC_SHA384
   * TLS_RSA_WITH_ARIA_128_GCM_SHA256
   * TLS_RSA_WITH_ARIA_256_GCM_SHA384
   * TLS_DH_RSA_WITH_ARIA_128_GCM_SHA256
   * TLS_DH_RSA_WITH_ARIA_256_GCM_SHA384
   * TLS_DH_DSS_WITH_ARIA_128_GCM_SHA256
   * TLS_DH_DSS_WITH_ARIA_256_GCM_SHA384
   * TLS_DH_anon_WITH_ARIA_128_GCM_SHA256
   * TLS_DH_anon_WITH_ARIA_256_GCM_SHA384
   * TLS_ECDH_ECDSA_WITH_ARIA_128_GCM_SHA256
   * TLS_ECDH_ECDSA_WITH_ARIA_256_GCM_SHA384

*   TLS_ECDH_RSA_WITH_ARIA_128_GCM_SHA256
*   TLS_ECDH_RSA_WITH_ARIA_256_GCM_SHA384
*   TLS_PSK_WITH_ARIA_128_CBC_SHA256
*   TLS_PSK_WITH_ARIA_256_CBC_SHA384
*   TLS_DHE_PSK_WITH_ARIA_128_CBC_SHA256
*   TLS_DHE_PSK_WITH_ARIA_256_CBC_SHA384
*   TLS_RSA_PSK_WITH_ARIA_128_CBC_SHA256
*   TLS_RSA_PSK_WITH_ARIA_256_CBC_SHA384
*   TLS_PSK_WITH_ARIA_128_GCM_SHA256
*   TLS_PSK_WITH_ARIA_256_GCM_SHA384
*   TLS_RSA_PSK_WITH_ARIA_128_GCM_SHA256
*   TLS_RSA_PSK_WITH_ARIA_256_GCM_SHA384
*   TLS_ECDHE_PSK_WITH_ARIA_128_CBC_SHA256
*   TLS_ECDHE_PSK_WITH_ARIA_256_CBC_SHA384
*   TLS_ECDHE_ECDSA_WITH_CAMELLIA_128_CBC_SHA256
*   TLS_ECDHE_ECDSA_WITH_CAMELLIA_256_CBC_SHA384
*   TLS_ECDH_ECDSA_WITH_CAMELLIA_128_CBC_SHA256
*   TLS_ECDH_ECDSA_WITH_CAMELLIA_256_CBC_SHA384
*   TLS_ECDHE_RSA_WITH_CAMELLIA_128_CBC_SHA256
*   TLS_ECDHE_RSA_WITH_CAMELLIA_256_CBC_SHA384
*   TLS_ECDH_RSA_WITH_CAMELLIA_128_CBC_SHA256
*   TLS_ECDH_RSA_WITH_CAMELLIA_256_CBC_SHA384
*   TLS_RSA_WITH_CAMELLIA_128_GCM_SHA256
*   TLS_RSA_WITH_CAMELLIA_256_GCM_SHA384
*   TLS_DH_RSA_WITH_CAMELLIA_128_GCM_SHA256
*   TLS_DH_RSA_WITH_CAMELLIA_256_GCM_SHA384
*   TLS_DH_DSS_WITH_CAMELLIA_128_GCM_SHA256

*   TLS_DH_DSS_WITH_CAMELLIA_256_GCM_SHA384
*   TLS_DH_anon_WITH_CAMELLIA_128_GCM_SHA256
*   TLS_DH_anon_WITH_CAMELLIA_256_GCM_SHA384
*   TLS_ECDH_ECDSA_WITH_CAMELLIA_128_GCM_SHA256
*   TLS_ECDH_ECDSA_WITH_CAMELLIA_256_GCM_SHA384
*   TLS_ECDH_RSA_WITH_CAMELLIA_128_GCM_SHA256
*   TLS_ECDH_RSA_WITH_CAMELLIA_256_GCM_SHA384
*   TLS_PSK_WITH_CAMELLIA_128_GCM_SHA256
*   TLS_PSK_WITH_CAMELLIA_256_GCM_SHA384
*   TLS_RSA_PSK_WITH_CAMELLIA_128_GCM_SHA256
*   TLS_RSA_PSK_WITH_CAMELLIA_256_GCM_SHA384
*   TLS_PSK_WITH_CAMELLIA_128_CBC_SHA256
*   TLS_PSK_WITH_CAMELLIA_256_CBC_SHA384
*   TLS_DHE_PSK_WITH_CAMELLIA_128_CBC_SHA256

* TLS_DHE_PSK_WITH_CAMELLIA_256_CBC_SHA384
   * TLS_RSA_PSK_WITH_CAMELLIA_128_CBC_SHA256
   * TLS_RSA_PSK_WITH_CAMELLIA_256_CBC_SHA384
   * TLS_ECDHE_PSK_WITH_CAMELLIA_128_CBC_SHA256
   * TLS_ECDHE_PSK_WITH_CAMELLIA_256_CBC_SHA384
   * TLS_RSA_WITH_AES_128_CCM
   * TLS_RSA_WITH_AES_256_CCM
   * TLS_RSA_WITH_AES_128_CCM_8
   * TLS_RSA_WITH_AES_256_CCM_8
   * TLS_PSK_WITH_AES_128_CCM
   * TLS_PSK_WITH_AES_256_CCM
   * TLS_PSK_WITH_AES_128_CCM_8
   * TLS_PSK_WITH_AES_256_CCM_8

   | Note: This list was assembled from the set of registered TLS
   | cipher suites at the time of writing. This list includes those
   | cipher suites that do not offer an ephemeral key exchange and
   | those that are based on the TLS null, stream, or block cipher
   | type (as defined in Section 6.2.3 of [TLS12]). Additional
   | cipher suites with these properties could be defined; these
   | would not be explicitly prohibited.

Appendix B. Changes from RFC 7540

   This revision includes a number of editorial updates, plus the
   following substantive changes:

   * Use of TLS 1.3 is defined based on RFC 8740, which this document
     obsoletes.

   * The priority scheme defined in RFC 7540 is deprecated.
     Definitions for the format of the PRIORITY frame and the priority
     fields in the HEADERS frame have been retained, plus the rules
     governing when PRIORITY frames can be sent and received, but the

     semantics of these fields is only described in RFC 7540. The
     priority signaling scheme from RFC 7540 was not successful. Using
     the simpler successor signaling [I-D.ietf-httpbis-priority] is
     recommended.

   * The HTTP/1.1 Upgrade mechanism is no longer specified in this
     document. It was never widely deployed, with plaintext HTTP/2

users choosing to use the prior-knowledge implementation instead.

   * The ranges of codepoints for settings and frame types that were
      reserved for "Experimental Use" are now available for general use.

Contributors

   The previous version of this document was authored by Mike Belshe and
   Roberto Peon.

Acknowledgements

   This document includes substantial input from the following
   individuals:

   * Adam Langley, Wan-Teh Chang, Jim Morrison, Mark Nottingham, Alyssa
      Wilk, Costin Manolache, William Chan, Vitaliy Lvin, Joe Chan, Adam
      Barth, Ryan Hamilton, Gavin Peters, Kent Alstad, Kevin Lindsay,
      Paul Amer, Fan Yang, and Jonathan Leighton (SPDY contributors).

   * Gabriel Montenegro and Willy Tarreau (Upgrade mechanism).

   * William Chan, Salvatore Loreto, Osama Mazahir, Gabriel Montenegro,
      Jitu Padhye, Roberto Peon, and Rob Trace (Flow control).

   * Mike Bishop (Extensibility).

   * Mark Nottingham, Julian Reschke, James Snell, Jeff Pinner, Mike
      Bishop, and Hervé Ruellan (Substantial editorial contributions).

   * Kari Hurtta, Tatsuhiro Tsujikawa, Greg Wilkins, Poul-Henning Kamp,
      and Jonathan Thackray.

   * Alexey Melnikov, who was an editor of this document in 2013.

   * David Benjamin, who was author of [RFC 8740](), the contents of which
      are integrated here.

   A substantial proportion of Martin's contribution was supported by
   Microsoft during his employment there.

The Japanese HTTP/2 community provided invaluable contributions, including a number of implementations as well as numerous technical and editorial contributions.

Authors' Addresses

Martin Thomson (editor)
Mozilla
Australia

Email: mt@lowentropy.net

Cory Benfield (editor)
Apple Inc.

Email: cbenfield@apple.com