HTTPbis Working Group
Internet-Draft
Intended status: Standards Track
Expires: October 25, 2014

M. Belshe
Twist
R. Peon
Google, Inc
M. Thomson, Ed.
Mozilla
April 23, 2014

Hypertext Transfer Protocol version 2
draft-ietf-httpbis-http2-12

Abstract

This specification describes an optimized expression of the syntax of the Hypertext Transfer Protocol (HTTP). 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 messages on the same connection. It also introduces unsolicited push of representations from servers to clients.

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

Editorial Note (To be removed by RFC Editor)

Discussion of this draft takes place on the HTTPBIS working group mailing list (ietf-http-wg@w3.org), which is archived at <http://lists.w3.org/Archives/Public/ietf-http-wg/>.

Working Group information can be found at <http://tools.ietf.org/wg/httpbis/>; that specific to HTTP/2 are at <http://http2.github.io/>.

The changes in this draft are summarized in Appendix A.

This version of HTTP/2, identified as "h2-12" or "h2c-12", is intended for implementation. An interoperability event will be conducted 2014-06-05, see <https://github.com/http2/wg_materials/blob/master/interim-14-06/agenda.md>.

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 http://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 October 25, 2014.

Table of Contents

1. Introduction

The Hypertext Transfer Protocol (HTTP) is a wildly successful protocol. However, the HTTP/1.1 message format ([HTTP-p1], Section 3) was designed to be implemented with the tools at hand in the 1990s, not modern Web application performance. As such it has several characteristics that have a negative overall effect on application performance today.

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

Furthermore, HTTP/1.1 header fields are often repetitive and verbose, which, in addition to generating more or larger network packets, can cause the small initial TCP congestion window to quickly fill. This can result in excessive latency when multiple requests are made on a single new TCP connection.

This document 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 designed to be 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 leads to better utilization of available network capacity.

Finally, this encapsulation also enables more scalable processing of messages through use of binary message framing.

2. HTTP/2 Protocol Overview

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

The basic protocol unit in HTTP/2 is a frame (Section 4.1). Each frame has a different type and 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 exchanged assigned to a single stream (Section 5). Streams are largely independent of each other, so a blocked or stalled request does not prevent progress on other requests.

Flow control and prioritization ensure that it is possible to properly 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 requests first.

HTTP/2 adds a new interaction mode, whereby a server can push responses to a client (Section 8.2). Server push allows a server to speculatively send a client data 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.

Frames that contain HTTP header fields are compressed (Section 4.3). HTTP requests can be highly redundant, so compression can reduce the size of requests and responses significantly.

HTTP/2 also supports HTTP Alternative Services (see [ALT-SVC]) using the ALTSVC frame type (Section 6.11), to allow servers more control over traffic to them.

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 framing (Section 4) and streams (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, the intent is not to define a completely generic framing layer. The framing and streams 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 initiating the HTTP/2 connection.

connection: A transport-level 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 bytes structured according to the frame type.

intermediary: A "proxy", "gateway" or other intermediary as defined in Section 2.3 of [HTTP-p1].

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 which did not initiate the HTTP/2 connection.

stream: A bi-directional flow of frames across a virtual channel within the HTTP/2 connection.

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

3. Starting HTTP/2

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

HTTP/2 uses the same "http" and "https" URI schemes used by HTTP/1.1. 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 "http" URIs is described in Section 3.2. Discovery for "https" URIs is 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 TLS [TLS12]. This identifier is used in the TLS application layer protocol negotiation extension [TLSALPN] field and any place that HTTP/2 over TLS is identified.

  When serialised into an ALPN protocol identifier (which is a sequence of octets), the HTTP/2 protocol identifier string is encoded using UTF-8 [UTF-8].

- The string "h2c" identifies the protocol where HTTP/2 is run over cleartext TCP. This identifier is used in the HTTP/1.1 Upgrade header field and any place that HTTP/2 over TCP is identified.

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

[[anchor3: RFC Editor's Note: please remove the remainder of this section prior to the publication of a final version of this document.]]

Only implementations of the final, published RFC can identify themselves as "h2" or "h2c". Until such an RFC exists, implementations MUST NOT identify themselves using these strings.

Examples and text throughout the rest of this document use "h2" as a matter of editorial convenience only. Implementations of draft

versions MUST NOT identify using this string.

Implementations of draft versions of the protocol MUST add the string "-" and the corresponding draft number to the identifier. For example, draft-ietf-httpbis-http2-11 over TLS is identified using the string "h2-11".

Non-compatible experiments that are based on these draft versions MUST append the string "-" and an experiment name to the identifier. For example, an experimental implementation of packet mood-based encoding based on draft-ietf-httpbis-http2-09 might identify itself as "h2-09-emo". Note that any label MUST conform to the "token" syntax defined in Section 3.2.6 of [HTTP-p1]. Experimenters are encouraged to coordinate their experiments on the ietf-http-wg@w3.org mailing list.

3.2. Starting HTTP/2 for "http" URIs

A client that makes a request to an "http" URI without prior knowledge about support for HTTP/2 uses the HTTP Upgrade mechanism (Section 6.7 of [HTTP-p1]). The client makes an HTTP/1.1 request that includes an Upgrade header field identifying HTTP/2 with the "h2c" token. The HTTP/1.1 request MUST include exactly one HTTP2-Settings (Section 3.2.1) header field.

For example:

```
GET /default.htm HTTP/1.1
Host: server.example.com
Connection: Upgrade, HTTP2-Settings
Upgrade: h2c
HTTP2-Settings: <base64url encoding of HTTP/2 SETTINGS payload>
```

Requests that contain an entity body MUST be sent in their entirety before the client can send HTTP/2 frames. This means that a large request entity can block the use of the connection until it is completely sent.

If concurrency of an initial request with subsequent requests is important, a small request can be used to perform the upgrade to HTTP/2, at the cost of an additional round-trip.

A server that does not support HTTP/2 can respond to the request as though the Upgrade header field were absent:

```
HTTP/1.1 200 OK
Content-Length: 243
Content-Type: text/html

...
```

A server that supports HTTP/2 can accept the upgrade with a 101 (Switching Protocols) response. After the empty line that terminates the 101 response, the server can begin sending HTTP/2 frames. These frames MUST include a response to the request that initiated the Upgrade.

```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: h2c

[ HTTP/2 connection ...
```

The first HTTP/2 frame sent by the server is a SETTINGS frame (Section 6.5). Upon receiving the 101 response, the client sends a connection preface (Section 3.5), which includes a SETTINGS frame.

The HTTP/1.1 request that is sent prior to upgrade is assigned stream identifier 1 and is assigned default priority values (Section 5.3.5). Stream 1 is implicitly half closed from the client toward the server, since the request is completed as an HTTP/1.1 request. After commencing the HTTP/2 connection, stream 1 is used for the response.

3.2.1. HTTP2-Settings Header Field

A request that upgrades from HTTP/1.1 to HTTP/2 MUST include exactly one "HTTP2-Settings" header field. The "HTTP2-Settings" header field is a hop-by-hop header field that includes parameters that govern the HTTP/2 connection, provided in anticipation of the server accepting the request to upgrade. A server MUST reject an attempt to upgrade if this header field is not present.

```
HTTP2-Settings    = token68
```

The content of the "HTTP2-Settings" header field is the payload of a SETTINGS frame (Section 6.5), encoded as a base64url string (that is, the URL- and filename-safe Base64 encoding described in Section 5 of [RFC4648], with any trailing '=' characters omitted). The ABNF [RFC5234] production for "token68" is defined in Section 2.1 of [HTTP-p7].

As a hop-by-hop header field, the "Connection" header field MUST include a value of "HTTP2-Settings" in addition to "Upgrade" when

upgrading to HTTP/2.

A server decodes and interprets these values as it would any other SETTINGS frame. Acknowledgement of the SETTINGS parameters (Section 6.5.3) is not necessary, since a 101 response serves as implicit acknowledgment. Providing these values in the Upgrade request ensures that the protocol does not require default values for the above SETTINGS parameters, and gives a client an opportunity to provide other parameters prior to receiving any frames from the server.

3.3. Starting HTTP/2 for "https" URIs

A client that makes a request to an "https" URI without prior knowledge about support for HTTP/2 uses TLS [TLS12] with the application layer protocol negotiation extension [TLSALPN].

Once TLS negotiation is complete, both the client and the server send a connection preface (Section 3.5).

3.4. 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 in an HTTP header field; the ALTSVC frame (Section 6.11) describes a similar mechanism in HTTP/2.

A client MAY immediately send HTTP/2 frames to a server that is known to support HTTP/2, after the connection preface (Section 3.5). A server can identify such a connection by the use of the "PRI" method in the connection preface. This only affects the resolution of "http" URIs; servers supporting HTTP/2 are required to support

protocol negotiation in TLS [TLSALPN] for "https" URIs.

Prior support for HTTP/2 is not a strong signal that a given server will support HTTP/2 for future connections. It is possible for server configurations to change; for configurations to differ between instances in clustered server; or network conditions to change.

3.5. HTTP/2 Connection Preface

Upon establishment of a TCP connection and determination that HTTP/2 will be used by both peers, each endpoint MUST send a connection preface as a final confirmation and to establish the initial SETTINGS parameters for the HTTP/2 connection.

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

0x505249202a20485454502f322e300d0a0d0a534d0d0a0d0a

(the string "PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n"). This sequence is followed by a SETTINGS frame (Section 6.5). The SETTINGS frame MAY be empty. The client sends the client connection preface immediately upon receipt of a 101 Switching Protocols response (indicating a successful upgrade), or as the first application data octets of a TLS connection. If starting an HTTP/2 connection with prior knowledge of server support for the protocol, the client connection preface is sent upon connection establishment.

> 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.

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 parameters 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 parameters established.

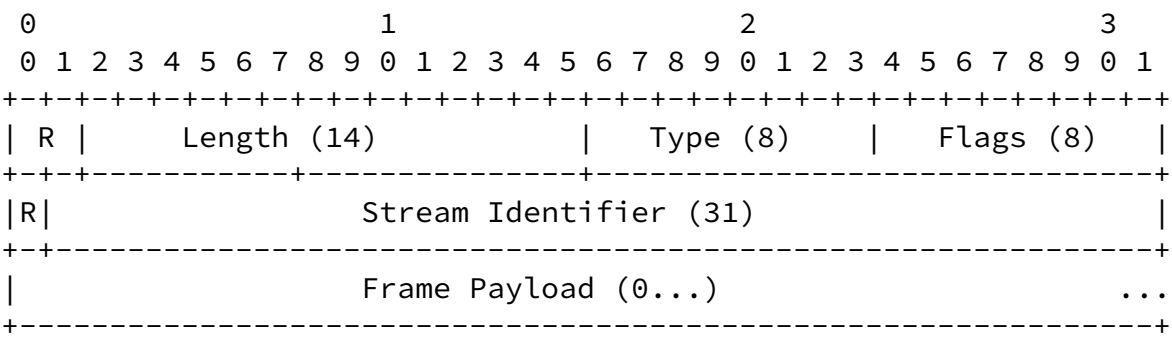
Clients and servers MUST terminate the TCP connection if either peer does not begin with a valid connection preface. A GOAWAY frame (Section 6.8) MAY be omitted if it is clear that the peer is not using HTTP/2.

4. HTTP Frames

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

4.1. Frame Format

All frames begin with an 8-octet header followed by a payload of between 0 and 16,383 octets.



Frame Header

The fields of the frame header are defined as:

R: A reserved 2-bit field. The semantics of these bits are undefined and the bits MUST remain unset (0) when sending and MUST be ignored when receiving.

Length: The length of the frame payload expressed as an unsigned 14-bit integer. The 8 octets of the frame header are not included in this value.

Type: The 8-bit type of the frame. The frame type determines how the remainder of the frame header and payload are interpreted. Implementations MUST treat the receipt of an unknown frame type (any frame types not defined in this document) as a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

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

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 (0) when sending.

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

Stream Identifier: A 31-bit stream identifier (see Section 5.1.1). The value 0 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 maximum size of a frame payload varies by frame type. The absolute maximum size of a frame payload is 2^14-1 (16,383) octets, meaning that the maximum frame size is 16,391 octets. All implementations SHOULD be capable of receiving and minimally processing frames up to this maximum size.

Certain frame types, such as PING (see Section 6.7), impose additional limits on the amount of payload data allowed. Likewise, additional size limits can be set by specific application uses (see

Section 9).

If a frame size exceeds any defined limit, or is too small to contain mandatory frame data, the endpoint MUST send a FRAME_SIZE_ERROR error. 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 header block (Section 4.3) (that is, HEADERS, PUSH_PROMISE, and CONTINUATION), SETTINGS, and any WINDOW_UPDATE frame with a stream identifier of 0.

4.3. Header Compression and Decompression

A header field in HTTP/2 is a name-value pair with one or more associated values. They are used within HTTP request and response messages as well as server push operations (see Section 8.2).

Header sets are collections of zero or more header fields. When transmitted over a connection, a header set is serialized into a header block using HTTP Header Compression [COMPRESSION]. The serialized header block is then divided into one or more octet sequences, called header block fragments, and transmitted within the payload of HEADERS (Section 6.2), PUSH_PROMISE (Section 6.6) or CONTINUATION (Section 6.10) frames.

HTTP Header Compression does not preserve the relative ordering of header fields. Header fields with multiple values are encoded into a single header field using a special delimiter; see Section 8.1.3.3.

The Cookie header field [COOKIE] is treated specially by the HTTP mapping; see Section 8.1.3.4.

A receiving endpoint reassembles the header block by concatenating its fragments, then decompresses the block to reconstruct the header set.

A complete header block 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.

Header compression is stateful, using a single compression context for the entire connection. Each header block is processed as a discrete unit. Header 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 MUST have the END_HEADERS flag set. The last frame in a sequence of PUSH_PROMISE or CONTINUATION frames MUST have the END_HEADERS flag set.

Header block fragments can only be sent as the 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 MUST reassemble header 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 header block.

5. Streams and Multiplexing

A "stream" is an independent, bi-directional 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.

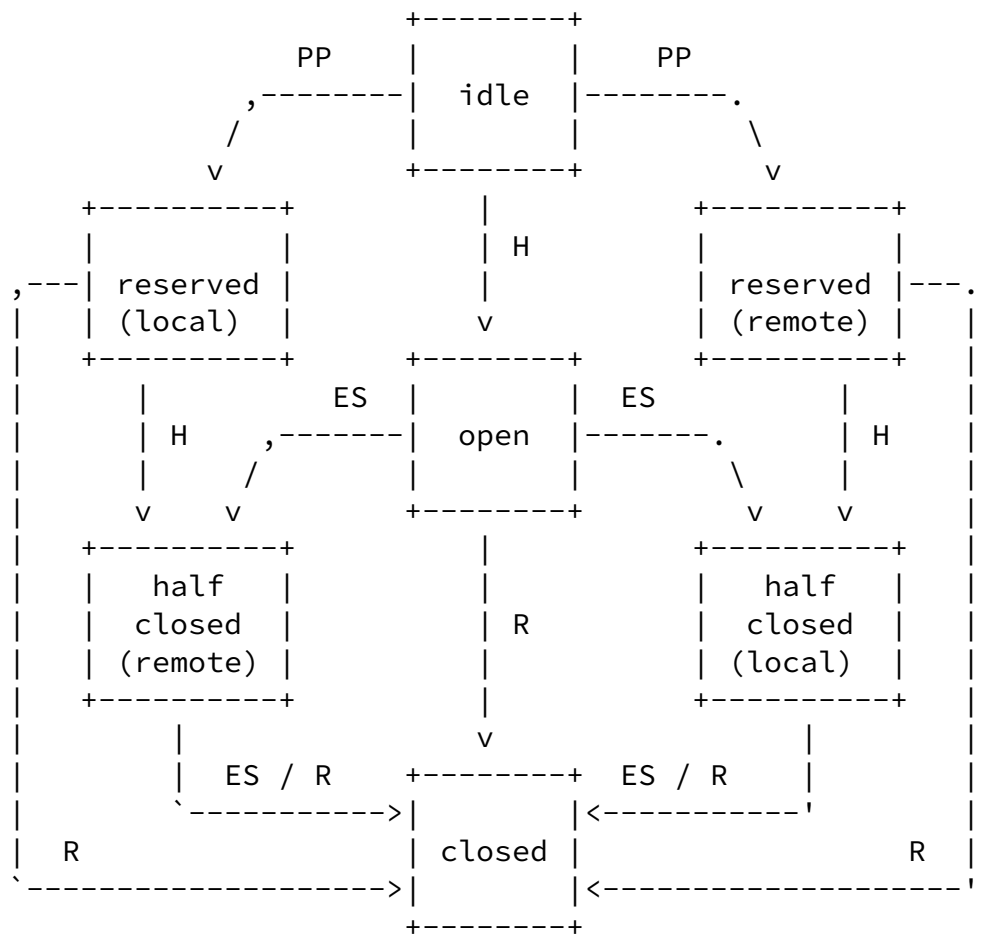
- Streams can be closed by either endpoint.

- The order in which frames are sent within a stream is significant. Recipients process frames in the order they are received.

- 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 1.



H: HEADERS frame (with implied CONTINUATIONs)
PP: PUSH_PROMISE frame (with implied CONTINUATIONs)
ES: END_STREAM flag
R: RST_STREAM frame

Figure 1: Stream States

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. In this state, no frames have been exchanged.

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 marks the associated stream for later use. The stream state for the reserved stream transitions to "reserved (local)".

* Receiving a PUSH_PROMISE frame marks the associated stream as reserved by the remote peer. The state of the stream becomes "reserved (remote)".

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 frames other than HEADERS or RST_STREAM in this state.

A PRIORITY frame MAY be received in this state. Receiving any frames other than RST_STREAM, or PRIORITY 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 MAY send a PRIORITY frame in this state to reprioritize the reserved stream. An endpoint MUST NOT send any other type of frame other than RST_STREAM or PRIORITY.

  Receiving any other type of frame other than HEADERS or RST_STREAM 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)". A HEADERS frame bearing an END_STREAM flag can be followed by CONTINUATION frames.

  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.

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. A HEADERS frame bearing an END_STREAM flag can be followed by CONTINUATION frames.

A receiver can ignore WINDOW_UPDATE or PRIORITY frames in this state. These frame types might arrive for a short period after a frame bearing the END_STREAM flag is sent.

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 it

performs flow control.

If an endpoint receives additional frames for a stream that is in this state, other than CONTINUATION frames, it MUST respond with a stream error (Section 5.4.2) of type STREAM_CLOSED.

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 endpoint MUST NOT send frames on a closed stream. An endpoint that receives any frame after receiving a RST_STREAM MUST treat that as a stream error (Section 5.4.2) of type STREAM_CLOSED. Similarly, an endpoint that receives any frames after receiving a DATA frame with the END_STREAM flag set, or any frames except a CONTINUATION frame after receiving a HEADERS frame with an END_STREAM flag set MUST treat that as a stream error (Section 5.4.2) of type STREAM_CLOSED.

WINDOW_UPDATE, PRIORITY, or RST_STREAM frames can be received in this state for a short period after a DATA or HEADERS frame containing an END_STREAM flag is sent. Until the remote peer receives and processes the frame bearing the END_STREAM flag, it

might send frame of any of these types. Endpoints MUST ignore WINDOW_UPDATE, PRIORITY, or RST_STREAM frames received in this state, though endpoints MAY choose to treat frames that arrive a significant time after sending END_STREAM as a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

If this state is reached as a result of sending a RST_STREAM frame, the peer that receives the RST_STREAM might have already sent - or enqueued for sending - frames on the stream that cannot be withdrawn. An endpoint MUST ignore frames that it receives on closed streams after it has sent a RST_STREAM frame. An endpoint MAY choose to limit the period over which it ignores frames and treat frames that arrive after this time as being in error.

Flow controlled frames (i.e., DATA) received after sending RST_STREAM are counted toward the connection flow control window. Even though these frames might be ignored, because they are sent before the sender receives the RST_STREAM, the sender will consider the frames to count against the flow control window.

An endpoint might receive a PUSH_PROMISE frame after it sends RST_STREAM. PUSH_PROMISE causes a stream to become "reserved"

even if the associated stream has been reset. Therefore, a RST_STREAM is needed to close an unwanted promised streams.

In the absence of more specific guidance elsewhere in this document, implementations SHOULD treat the receipt of a message that is not expressly permitted in the description of a state as a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

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 zero MUST NOT be used to establish a new stream.

HTTP/1.1 requests that are upgraded to HTTP/2 (see Section 3.2) are responded to with a stream identifier of one (0x1). After the

upgrade completes, stream 0x1 is "half closed (local)" to the client. Therefore, stream 0x1 cannot be selected as a new stream identifier by a client that upgrades from HTTP/1.1.

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.

The first use of a new stream identifier implicitly closes all streams in the "idle" state that might have been initiated by that peer with a lower-valued stream identifier. For example, if a client sends a HEADERS frame on stream 7 without ever sending a frame on stream 5, then stream 5 transitions to the "closed" state when the first frame for stream 7 is sent or received.

Stream identifiers cannot be reused. Long-lived connections can result in 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.

5.1.2. Stream Concurrency

A peer can limit the number of concurrently active streams using the SETTINGS_MAX_CONCURRENT_STREAMS parameters 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. Endpoints MUST NOT exceed the limit set by their peer.

Streams that are in the "open" state, or 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 (see Section 6.5.2).

An endpoint that receives a HEADERS frame that causes their

advertised concurrent stream limit to be exceeded MUST treat this as a stream error (Section 5.4.2).

Streams in either of the "reserved" states do not count as open.

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 type.

5.2.1. Flow Control Principles

HTTP/2 stream flow control aims to allow for future improvements to flow control algorithms without requiring protocol changes. Flow control in HTTP/2 has the following characteristics:

1. Flow control is hop-by-hop, not end-to-end.

2. Flow control is based on window update frames. Receivers advertise how many bytes 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 bytes 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 standardizes only the format of the WINDOW_UPDATE frame (Section 6.9). This 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 managing how requests and responses are sent based on priority; 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 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, incrementing the available space when new data is received. Sending data is always subject to the flow control window advertised by the receiver.

Deployments with constrained resources (for example, memory) MAY 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 [RFC1323]).

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 available to HTTP/2. However, flow control can ensure that constrained resources are protected without any reduction in connection utilization.

5.3. Stream priority

A client can assign a priority for a new stream by including prioritization information in the HEADERS frame (Section 6.2) that opens the stream. For an existing stream, the PRIORITY frame (Section 6.3) can be used to change the priority.

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

Streams can be prioritized by marking them as dependent on the completion of other streams (Section 5.3.1). Each dependency is assigned a relative weight, a number that is used to determine the relative proportion of available resources that are assigned to streams dependent on the same stream.

Explicitly setting the priority for a stream is input to a prioritization process. It does not guarantee any particular processing or transmission order for the stream relative to any other stream. An endpoint cannot force a peer to process concurrent streams in a particular order using priority. Expressing priority is therefore only ever a suggestion.

Prioritization information can be specified explicitly for streams as they are created using the HEADERS frame, or changed using the PRIORITY frame. Providing prioritization information is optional, so default values are used if no explicit indicator is provided (Section 5.3.5).

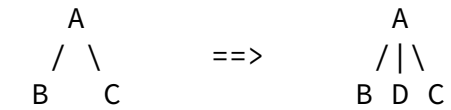
5.3.1. Stream Dependencies

Each stream can be given an explicit dependency on another stream. Including a dependency expresses a preference to allocate resources to the identified stream rather than to the dependent stream.

A stream that is not dependent on any other stream can given a stream dependency of 0x0.

When assigning a dependency on another stream, by default, the stream is added as a new dependency of the stream it depends on. For example, if streams B and C are dependent on stream A, and if stream D is created with a dependency on stream A, this results in a

dependency order of A followed by B, C, and D.



Example of Default Dependency Creation

An exclusive flag allows for the insertion of a new level of dependencies. The exclusive flag causes the stream to become the sole dependency of the stream it depends on, causing other dependencies to become dependencies of the stream. In the previous example, if stream D is created with an exclusive dependency on stream A, this results in a dependency order of A followed by D followed by B and C.

Example of Exclusive Dependency Creation

Inside the dependency tree, a dependent stream SHOULD only be allocated resources if the streams that it depends on are either closed, or it is not possible to make progress on them.

5.3.2. Dependency Weighting

Each dependency is allocated an integer weight between 1 to 256 (inclusive).

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

A stream MUST NOT depend on itself. An endpoint MAY either treat this as a stream error (Section 5.4.2) of type PROTOCOL_ERROR, or assign default priority values (Section 5.3.5) to the stream.

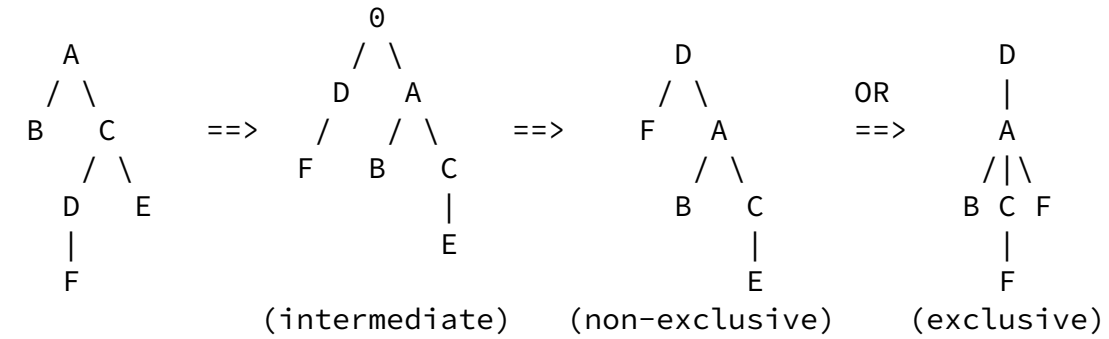
5.3.3. Reprioritization

Stream priorities are changed using the PRIORITY frame. Setting a dependency causes a stream to become dependent on the identified stream.

All streams that are dependent on a reprioritized stream move with it. Setting a dependency with the exclusive flag for a reprioritized stream moves all the dependencies of the stream it depends on to become dependencies of the reprioritized stream.

If a stream is made dependent on one of its own dependencies, the formerly dependent stream is first moved to be depedent on the reprioritized streams previous dependency, retaining its weight.

For example, for an original dependency tree where B and C depend on A, D and E depend on C, and F depends on D; if A is made dependent on D, then D takes the place of A with A dependent on D and all other dependency relationships staying the same.



Example of Dependency Reordering

5.3.4. Prioritization State Management

When a stream is removed from the dependency tree, its dependencies can be moved to become dependent on the stream the closed stream depends on. The weights of new dependencies SHOULD be assigned by distributing the weight of the dependency of the closed stream proportionally based on the weights of its dependencies.

Streams that are removed from the dependency tree cause some

prioritization information to be lost. Resources are shared between streams that depend on the same stream, which means that if a stream in that set closes or becomes blocked, any spare capacity allocated to a stream is distributed to the immediate neighbors of the stream. However, if the common dependency is removed from the tree, those streams share resources with streams at the next highest level. For example, assume streams A and B share a dependency, and C and D both depend on A. Prior to the removal of A, if stream A and D are unable to proceed, then C receives all the resources dedicated to A. If A is removed from the tree, the weight of A is divided equally between D and E, which results in stream C receiving a reduced proportion of resources (one third, rather than one half).

It is possible for a stream to become closed while prioritization information that creates a dependency on that stream is in transit. If a stream identified in a dependency has been closed and any associated priority information destroyed then the dependent stream is instead assigned a default priority. This potentially creates suboptimal prioritization, since the stream can be given an effective priority that is higher than expressed by a peer.

To avoid these problems, endpoints SHOULD maintain prioritization state for closed streams for a period after streams close.

An endpoint SHOULD retain stream prioritization state for a period after streams become closed. The longer state is retained, the lower the chance that streams are assigned incorrect or default priority values.

This could create a large state burden for an endpoint, so this state MAY be limited. An endpoint MAY apply a fixed upper limit on the number of closed streams for which prioritization state is tracked to limit state exposure. The amount of additional state an endpoint maintains could be dependent on load; under high load, prioritization state can be discarded to limit resource commitments. In extreme cases, an endpoint could even discard prioritization state for active or reserved streams. If a fixed limit is applied, endpoints SHOULD maintain state for at least as many streams as allowed by their setting for SETTINGS_MAX_CONCURRENT_STREAMS.

An endpoint receiving a PRIORITY frame that changes the priority of a

closed stream SHOULD alter the dependencies of the streams that depend on it, if it has retained enough state to do so.

5.3.5. Default Priorities

Providing priority information is optional. Streams are assigned a dependency on stream 0x0. Pushed streams (Section 8.2) initially depend on their associated stream. In both cases, streams are assigned a default weight of 16.

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.

5.4.1. Connection Error Handling

A connection error is any error which prevents further processing of the framing layer, or which 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, 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, as long as circumstances permit it.

5.4.2. Stream Error Handling

A stream error is an error related to a specific stream identifier 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 header compression (Section 4.3)).

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.

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

5.4.3. Connection Termination

If the TCP connection is torn down while streams remain in open or half closed states, then the endpoint MUST assume that those streams were abnormally interrupted and could be incomplete.

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 either in the establishment and management of the connection as a whole, or of individual streams.

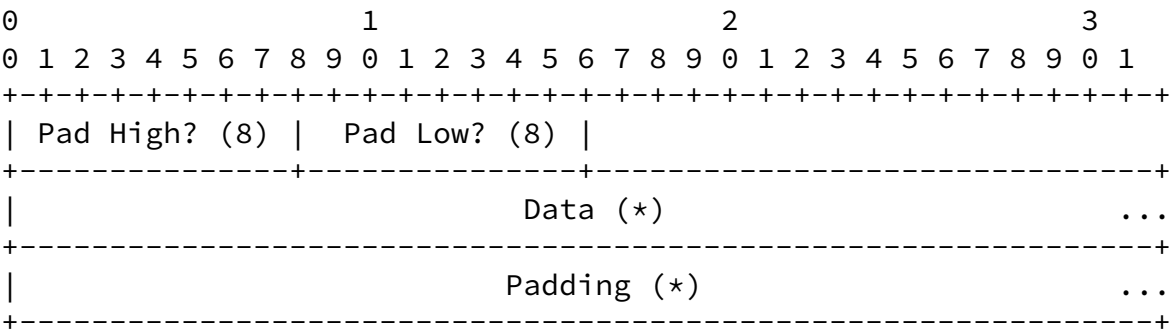
The transmission of specific frame types can alter the state of a

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 payloads.

DATA frames MAY also contain arbitrary padding. Padding can be added to DATA frames to hide the size of messages.



DATA Frame Payload

The DATA frame contains the following fields:

Pad High: An 8-bit field containing an amount of padding in units of 256 octets. This field is optional and is only present if the PAD_HIGH flag is set. This field, in combination with Pad Low, determines how much padding there is on a frame.

Pad Low: An 8-bit field containing an amount of padding in units of single octets. This field is optional and is only present if the PAD_LOW flag is set. This field, in combination with Pad High, determines how much padding there is on a frame.

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 and ignored when receiving.

The DATA frame defines the following flags:

END_STREAM (0x1): Bit 1 being set 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).

END_SEGMENT (0x2): Bit 2 being set indicates that this frame is the last for the current segment. Intermediaries MUST NOT coalesce frames across a segment boundary and MUST preserve segment boundaries when forwarding frames.

PAD_LOW (0x8): Bit 4 being set indicates that the Pad Low field is present.

PAD_HIGH (0x10): Bit 5 being set indicates that the Pad High field is present. This bit MUST NOT be set unless the PAD_LOW flag is also set. Endpoints that receive a frame with PAD_HIGH set and PAD_LOW cleared MUST treat this as a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

COMPRESSED (0x20): Bit 6 being set indicates that the data in the frame has been compressed with GZIP compression ([GZIP]).

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 optionally compressed using GZip compression [GZIP]. Each frame is individually compressed; the state of the compressor is reset for each frame.

An endpoint MUST NOT send a DATA frame with the COMPRESSED flag set unless the SETTINGS_COMPRESS_DATA setting is enabled, that is, set to

1. An endpoint that has not enabled DATA frame compression MUST treat the receipt of a DATA frame with the COMPRESSED flag set as 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)" states. Padding is included in flow control. 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.

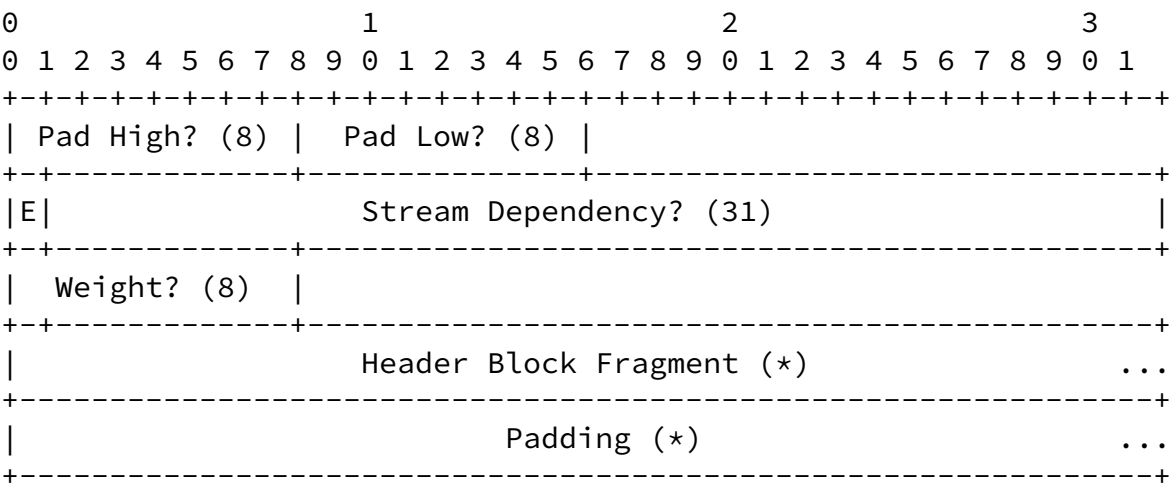
The total number of padding octets is determined by multiplying the value of the Pad High field by 256 and adding the value of the Pad Low field. Both Pad High and Pad Low fields assume a value of zero if absent. If the length of the padding is greater than the length of the remainder of the frame payload, 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 Low field with a value of zero.

Use of padding is a security feature; as such, its use demands some care, see Section 10.7.

6.2. HEADERS

The HEADERS frame (type=0x1) carries name-value pairs. It is used to open a stream (Section 5.1). HEADERS frames can be sent on a stream in the "open" or "half closed (remote)" states.



HEADERS Frame Payload

The HEADERS frame payload has the following fields:

Pad High: Padding size high bits. This field is only present if the PAD_HIGH flag is set.

Pad Low: Padding size low bits. This field is only present if the PAD_LOW flag is set.

E: A single bit flag indicates that the stream dependency is exclusive, see Section 5.3. This field is optional and is only present if the PRIORITY flag is set.

Stream Dependency: A 31-bit stream identifier for the stream that this stream depends on, see Section 5.3. This field is optional and is only present if the PRIORITY flag is set.

Weight: An 8-bit weight for the stream, see Section 5.3. Add one to the value to obtain a weight between 1 and 256. This field is optional and is only present if the PRIORITY flag is set.

Header Block Fragment: A header block fragment (Section 4.3).

Padding: Padding octets.

The HEADERS frame defines the following flags:

END_STREAM (0x1): Bit 1 being set indicates that the header block (Section 4.3) is the last that the endpoint will send for the identified stream. Setting this flag causes the stream to enter one of "half closed" states (Section 5.1).

A HEADERS frame that is followed by CONTINUATION frames carries the END_STREAM flag that signals the end of a stream. A CONTINUATION frame cannot be used to terminate a stream.

END_SEGMENT (0x2): Bit 2 being set indicates that this frame is the last for the current segment. Intermediaries MUST NOT coalesce frames across a segment boundary and MUST preserve segment boundaries when forwarding frames.

END_HEADERS (0x4): Bit 3 being set indicates that this frame contains an entire header 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.

PAD_LOW (0x8): Bit 4 being set indicates that the Pad Low field is present.

PAD_HIGH (0x10): Bit 5 being set indicates that the Pad High field is present. This bit MUST NOT be set unless the PAD_LOW flag is also set. Endpoints that receive a frame with PAD_HIGH set and PAD_LOW cleared MUST treat this as a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

PRIORITY (0x20): Bit 6 being set indicates that the Exclusive Flag (E), Stream Dependency, and Weight fields are present; see Section 5.3.

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

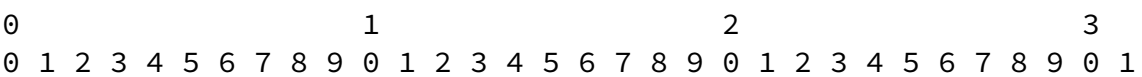
HEADERS frames MUST be associated with a stream. If a HEADERS frame is received whose stream identifier field is 0x0, the recipient MUST respond with a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

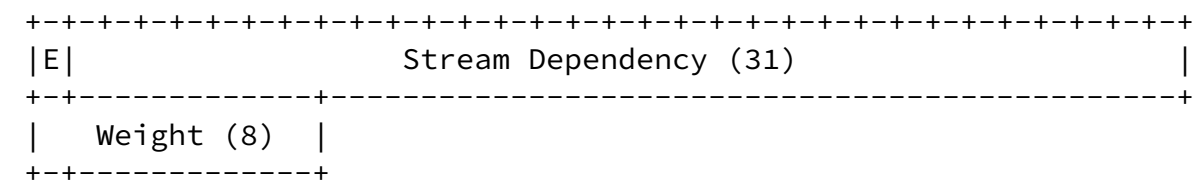
The HEADERS frame changes the connection state as described in Section 4.3.

The HEADERS frame includes optional padding. Padding fields and flags are identical to those defined for DATA frames (Section 6.1).

6.3. PRIORITY

The PRIORITY frame (type=0x2) specifies the sender-advised priority of a stream (Section 5.3). It can be sent at any time for an existing stream. This enables reprioritization of existing streams.





PRIORITY Frame Payload

The payload of a PRIORITY frame contains the following fields:

E: A single bit flag indicates that the stream dependency is exclusive, see Section 5.3.

Stream Dependency: A 31-bit stream identifier for the stream that this stream depends on, see Section 5.3.

Weight: An 8-bit weight for the identified stream dependency, see Section 5.3. Add one to the value to obtain a weight between 1 and 256.

The PRIORITY frame does not define any flags.

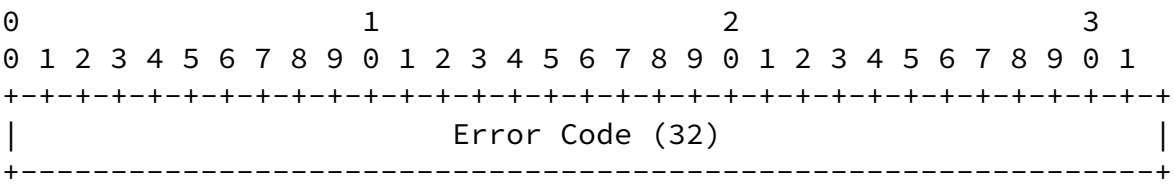
The PRIORITY frame is associated with an existing stream. If a PRIORITY frame is received with a stream identifier of 0x0, the recipient MUST respond with a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

The PRIORITY frame can be sent on a stream in any of the "reserved (remote)", "open", "half-closed (local)", or "half closed (remote)" states, though it cannot be sent between consecutive frames that comprise a single header block (Section 4.3). Note that this frame could arrive after processing or frame sending has completed, which would cause it to have no effect. For a stream that is in the "half closed (remote)" state, this frame can only affect processing of the stream and not frame transmission.

6.4. RST_STREAM

The RST_STREAM frame (type=0x3) allows for abnormal termination of a

stream. When sent by the initiator of a stream, it indicates that they wish to cancel the stream or that an error condition has occurred. When sent by the receiver of a stream, it indicates that either the receiver is rejecting the stream, requesting that the stream be cancelled or that an error condition has occurred.



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. 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.

6.5. SETTINGS

The SETTINGS frame (type=0x4) conveys configuration parameters (such as preferences and constraints on peer behavior) that affect how

endpoints communicate, and is also used to acknowledge the receipt of those parameters. Individually, a SETTINGS parameter can also be referred to as a "setting".

SETTINGS parameters are not negotiated; they describe characteristics of the sending peer, which are used by the receiving peer. Different values for the same parameter 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 parameters defined by this specification.

Each parameter in a SETTINGS frame replaces any existing value for that parameter. Parameters 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 its parameters. Therefore, the value of a SETTINGS parameter is the last value that is seen by a receiver.

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

ACK (0x1): Bit 1 being set indicates that this frame acknowledges receipt and application of the peer's SETTINGS frame. When this bit is set, the 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 info, see Settings Synchronization (Section 6.5.3).

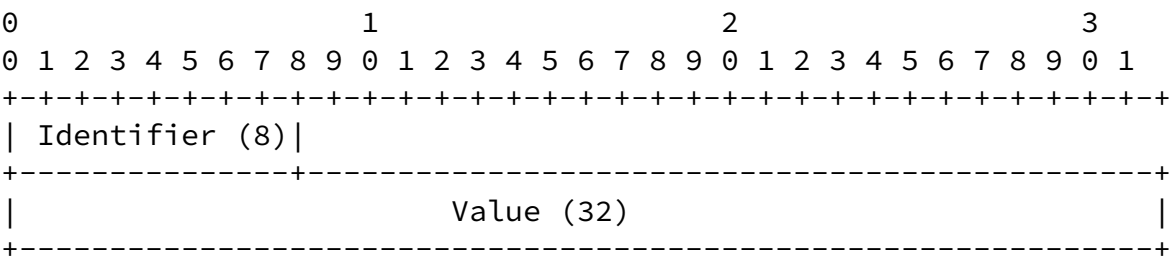
SETTINGS frames always apply to a connection, never a single stream. The stream identifier for a SETTINGS frame MUST be zero. If an endpoint receives a SETTINGS frame whose stream identifier field is anything other than 0x0, the endpoint MUST respond with a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

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.

6.5.1. SETTINGS Format

The payload of a SETTINGS frame consists of zero or more parameters, each consisting of an unsigned 8-bit identifier and an unsigned 32-bit value.



Setting Format

6.5.2. Defined SETTINGS Parameters

The following parameters are defined:

SETTINGS_HEADER_TABLE_SIZE (1): Allows the sender to inform the remote endpoint of the size of the header compression table used to decode header blocks. The encoder can reduce this size by using signaling specific to the header compression format inside a header block. The initial value is 4,096 bytes.

SETTINGS_ENABLE_PUSH (2): This setting can be use to disable server push (Section 8.2). An endpoint MUST NOT send a PUSH_PROMISE frame if it receives this parameter set to a value of 0. An endpoint 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 is 1, which indicates that 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.

SETTINGS_MAX_CONCURRENT_STREAMS (3): 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 could be preferable.

SETTINGS_INITIAL_WINDOW_SIZE (4): Indicates the sender's initial window size (in bytes) for stream level flow control. The initial value is 65,535.

This setting affects the window size of all streams, including existing 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_COMPRESS_DATA (5): This setting is used to enable GZip compression of DATA frames.

A value of 1 indicates that DATA frames MAY be compressed. A value of 0 indicates that compression is not permitted. The initial value is 0.

Values other than 0 or 1 are invalid. An endpoint MUST treat the receipt of any other value as a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

An endpoint that receives a SETTINGS frame with any other identifier MUST treat this as a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

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 the communicated 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 parameters as soon as possible upon receipt.

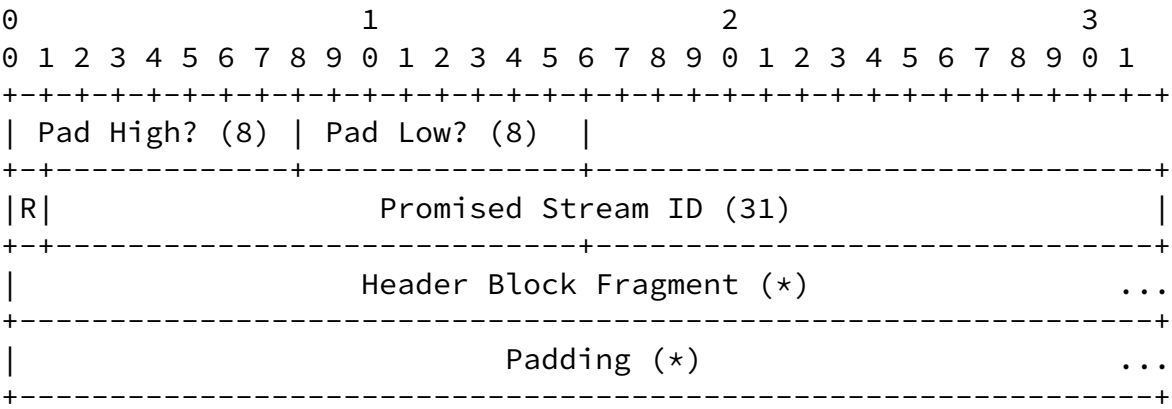
The values in the SETTINGS frame MUST be applied in the order they appear, with no other frame processing between values. Once all values have been applied, the recipient MUST immediately emit a SETTINGS frame with the ACK flag set. Upon receiving a SETTINGS frame with the ACK flag set, the sender of the altered parameters can rely upon their application.

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 set of headers that provide additional context for the stream. Section 8.2 contains a thorough description of the use of PUSH_PROMISE frames.

PUSH_PROMISE MUST NOT be sent if the SETTINGS_ENABLE_PUSH setting of the peer endpoint is set to 0.



PUSH_PROMISE Payload Format

The PUSH_PROMISE frame payload has the following fields:

Pad High: Padding size high bits. This field is only present if the PAD_HIGH flag is set.

Pad Low: Padding size low bits. This field is only present if the PAD_LOW flag is set.

R: A single reserved bit.

Promised Stream ID: This unsigned 31-bit integer identifies the stream the endpoint intends to start sending frames for. The promised stream identifier MUST be a valid choice for the next stream sent by the sender (see new stream identifier (Section 5.1.1)).

Header Block Fragment: A header block fragment (Section 4.3) containing request header fields.

Padding: Padding octets.

The PUSH_PROMISE frame defines the following flags:

END_HEADERS (0x4): Bit 3 being set indicates that this frame contains an entire header 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.

PAD_LOW (0x8): Bit 4 being set indicates that the Pad Low field is present.

PAD_HIGH (0x10): Bit 5 being set indicates that the Pad High field is present. This bit MUST NOT be set unless the PAD_LOW flag is also set. Endpoints that receive a frame with PAD_HIGH set and PAD_LOW cleared MUST treat this as a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

PUSH_PROMISE frames MUST be associated with an existing, peer-initiated stream. 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 order promised. The PUSH_PROMISE only reserves stream identifiers for later use.

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.

The PUSH_PROMISE frame modifies the connection state as defined in Section 4.3.

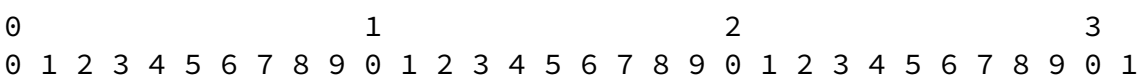
A PUSH_PROMISE frame modifies the connection state in two ways. The inclusion of a header block (Section 4.3) potentially modifies the state maintained for header compression. PUSH_PROMISE also reserves a stream for later use, causing the promised stream to enter the "reserved" state. A sender MUST NOT 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).

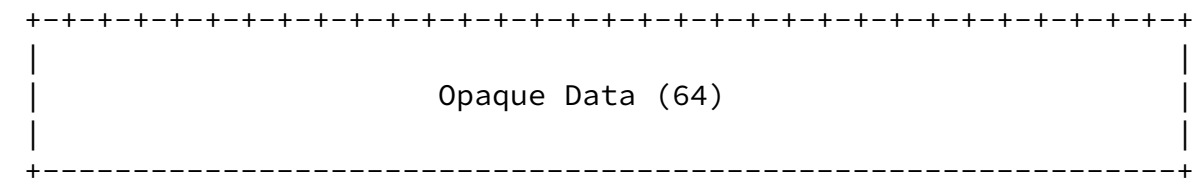
Since PUSH_PROMISE reserves a stream, ignoring a PUSH_PROMISE frame causes the stream state to become indeterminate. A receiver MUST treat the receipt of a PUSH_PROMISE on a stream that is neither "open" nor "half-closed (local)" as a connection error (Section 5.4.1) of type PROTOCOL_ERROR. Similarly, a receiver MUST treat the receipt of a PUSH_PROMISE that promises an illegal stream identifier (Section 5.1.1) (that is, an identifier for a stream that is not currently in the "idle" state) as a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

The PUSH_PROMISE frame includes optional 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.





PING Payload Format

In addition to the frame header, PING frames MUST contain 8 octets of

data in the payload. A sender can include any value it chooses and use those bytes 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 payload. PING responses SHOULD be given higher priority than any other frame.

The PING frame defines the following flags:

ACK (0x1): Bit 1 being set 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

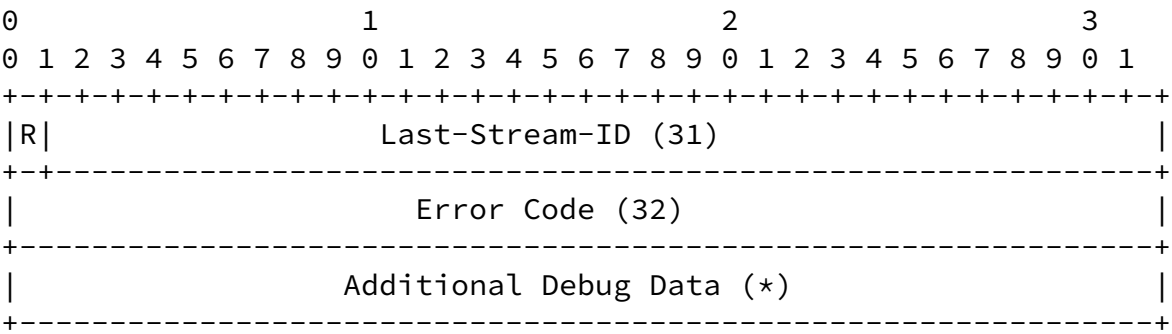
The GOAWAY frame (type=0x7) informs the remote peer to stop creating streams on this connection. GOAWAY can be sent by either the client or the server. Once sent, the sender will ignore frames sent on new streams for the remainder of the connection. Receivers of a GOAWAY frame MUST NOT open additional streams on the connection, although a new connection can be established for new streams. The purpose of this frame is to allow an endpoint to gracefully stop accepting new

streams (perhaps for a reboot or maintenance), while still finishing processing of previously established streams.

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 stream which was processed on the sending endpoint in this connection. If the receiver of the GOAWAY used streams that are newer than the indicated stream identifier, they were not processed by the sender and the receiver may treat the streams as though they had never been created at all (hence the receiver may want to re-create the streams later on a new connection).

Endpoints SHOULD always send a GOAWAY frame before closing a connection so that the remote 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 where it stopped working. An endpoint might choose to close a connection without sending GOAWAY for misbehaving peers.



GOAWAY Payload Format

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 has received frames and might have taken some 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, this value 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, re-attempting 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 an condition that requires immediate termination of the connection. The last stream identifier from the last GOAWAY frame received applies.

After sending a GOAWAY frame, the sender can discard frames for streams 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 header 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 header 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 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

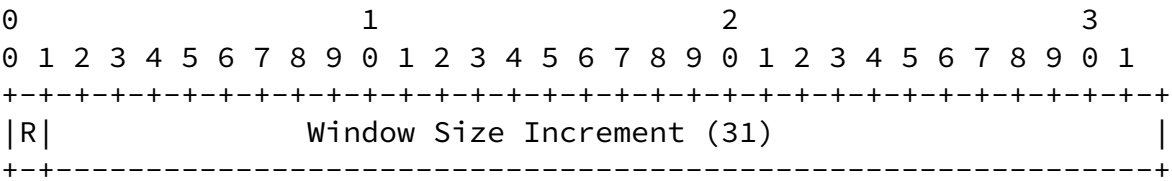
The WINDOW_UPDATE frame (type=0x8) is used to implement flow control; see Section 5.2 for an overview.

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 frame. 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.



WINDOW_UPDATE Payload Format

The payload of a WINDOW_UPDATE frame is one reserved bit, plus an unsigned 31-bit integer indicating the number of bytes 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 (0x7fffffff) bytes.

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.

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. Since the sender counts the frame toward the flow control window, if the receiver does not, the flow control window at sender and receiver can become different.

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 bytes of data the sender is permitted to transmit; as such, its size is a measure of the buffering capability 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 (for

example, an empty data frame) MAY be sent if there is no available space in either flow control window.

For flow control calculations, the 8 byte 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 bytes. 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 the error code of FLOW_CONTROL_ERROR code; for the connection, a GOAWAY frame with a FLOW_CONTROL_ERROR code.

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.

A sender that is unable to send data on a stream due to either flow control window being zero or lower MAY send a BLOCKED frame (Section 6.12) in order to inform the receiver of a potential flow control problem.

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 bytes. The connection flow control window is 65,535 bytes. 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 initial size cannot be changed.

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.

A SETTINGS frame can alter the initial flow control window size for all current streams. 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 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.

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 60KB immediately on connection establishment, and the server sets the initial window size to be 16KB, the client will recalculate the available flow control window to be -44KB 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.

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 has two options for handling streams that exceed flow control limits:

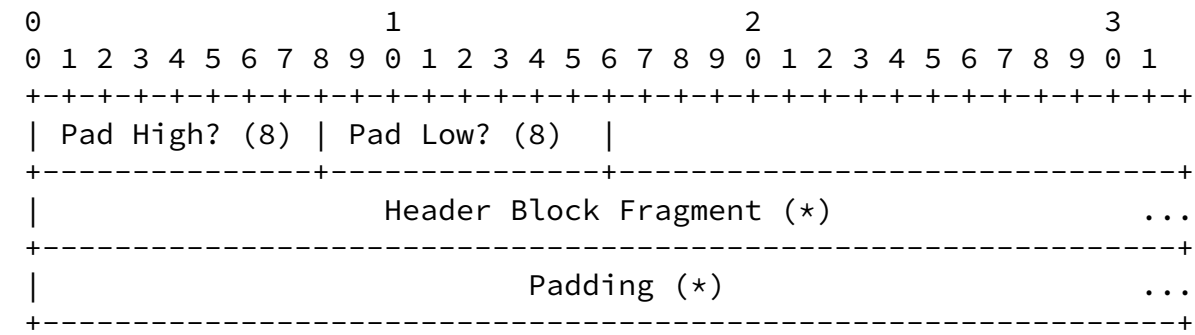
1. The receiver can immediately send RST_STREAM with FLOW_CONTROL_ERROR error code for the affected streams.

2. The receiver can accept the streams and tolerate the resulting head of line blocking, sending WINDOW_UPDATE frames as it

consumes data.

6.10. CONTINUATION

The CONTINUATION frame (type=0x9) is used to continue a sequence of header block fragments (Section 4.3). Any number of CONTINUATION frames can be sent on an existing stream, 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.



CONTINUATION Frame Payload

The CONTINUATION frame payload has the following fields:

Pad High: Padding size high bits. This field is only present if the PAD_HIGH flag is set.

Pad Low: Padding size low bits. This field is only present if the PAD_LOW flag is set.

Header Block Fragment: A header block fragment (Section 4.3).

Padding: Padding octets.

The CONTINUATION frame defines the following flags:

END_HEADERS (0x4): Bit 3 being set indicates that this frame ends a header 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.

PAD_LOW (0x8): Bit 4 being set indicates that the Pad Low field is present.

PAD_HIGH (0x10): Bit 5 being set indicates that the Pad High field is present. This bit MUST NOT be set unless the PAD_LOW flag is also set. Endpoints that receive a frame with PAD_HIGH set and PAD_LOW cleared MUST treat this as a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

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

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.

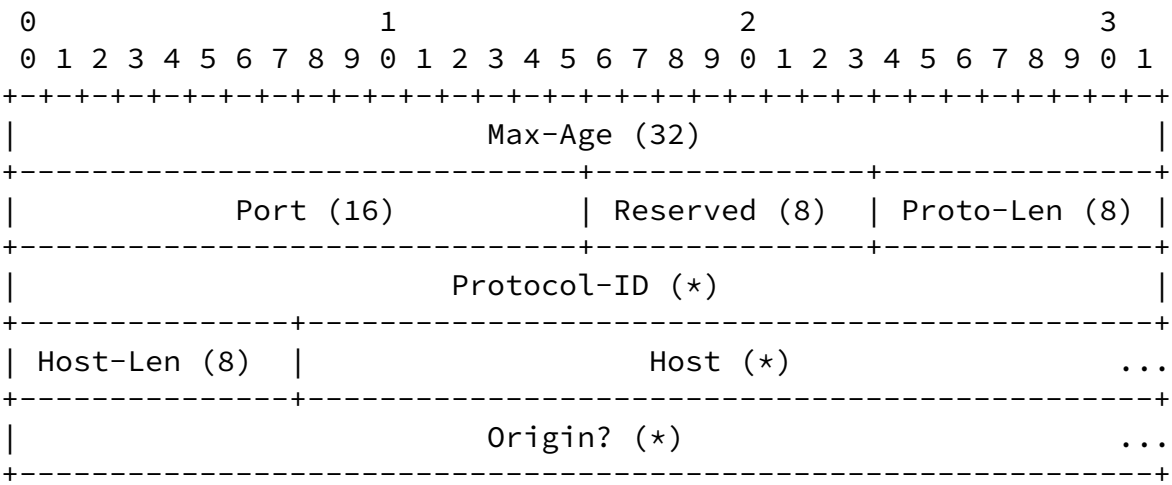
The CONTINUATION frame includes optional padding. Padding fields and flags are identical to those defined for DATA frames (Section 6.1).

6.11. ALTSVC

The ALTSVC frame (type=0xA) advertises the availability of an alternative service to the client. It can be sent at any time for an existing client-initiated stream or stream 0, and is intended to allow servers to load balance or otherwise segment traffic; see [ALT-SVC] for details (in particular, Section 2.4, which outlines client handling of alternative services).

An ALTSVC frame on a client-initiated stream indicates that the conveyed alternative service is associated with the origin of that stream.

An ALTSVC frame on stream 0 indicates that the conveyed alternative service is associated with the origin contained in the Origin field of the frame. An association with an origin that the client does not consider authoritative for the current connection MUST be ignored.



ALTSVC Frame Payload

The ALTSVC frame contains the following fields:

Max-Age: An unsigned, 32-bit integer indicating the freshness lifetime of the alternative service association, as per [ALT-SVC], Section 2.2.

Port: An unsigned, 16-bit integer indicating the port that the alternative service is available upon.

Reserved: For future use. Senders MUST set these bits to '0', and

recipients MUST ignore them.

Proto-Len: An unsigned, 8-bit integer indicating the length, in octets, of the Protocol-ID field.

Protocol-ID: A sequence of bytes (length determined by "Proto-Len") containing the ALPN protocol identifier of the alternative service.

Host-Len: An unsigned, 8-bit integer indicating the length, in octets, of the Host field.

Host: A sequence of characters (length determined by "Host-Len") containing an ASCII string indicating the host that the alternative service is available upon. An internationalized domain name [IDNA] MUST be expressed using A-labels.

Origin: An optional sequence of characters (length determined by subtracting the length of all preceding fields from the frame length) containing the ASCII serialisation of an origin ([RFC6454], Section 6.2) that the alternate service is applicable to.

The ALTSVC frame does not define any flags.

The ALTSVC frame is intended for receipt by clients; a server that receives an ALTSVC frame MUST treat it as a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

The ALTSVC frame is processed hop-by-hop. An intermediary MUST NOT forward ALTSVC frames, though it can use the information contained in ALTSVC frames in forming new ALTSVC frames to send to its own clients.

6.12. BLOCKED

The BLOCKED frame (type=0xB) indicates that the sender is unable to send data due to a closed flow control window.

[[anchor12: The BLOCKED frame is included in this draft version to facilitate experimentation. If the results of the experiment do not provide positive feedback, it could be removed.]]

The BLOCKED frame is used to provide feedback about the performance of flow control for the purposes of performance tuning and debugging. The BLOCKED frame can be sent by a peer when flow controlled data cannot be sent due to the connection- or stream-level flow control. This frame MUST NOT be sent if there are other reasons preventing data from being sent, either a lack of available data, or the underlying transport being blocked.

The BLOCKED frame is sent on the stream that is blocked, that is, the stream with a non-positive number of bytes available in the flow control window. A BLOCKED frame can be sent on stream 0x0 to indicate that connection-level flow control is blocked.

An endpoint MUST NOT send any subsequent BLOCKED frames until the affected flow control window becomes positive. This means that WINDOW_UPDATE frames are received or SETTINGS_INITIAL_WINDOW_SIZE is increased before more BLOCKED frames can be sent.

The BLOCKED frame defines no flags and contains no payload. A receiver MUST treat the receipt of a BLOCKED frame with a payload as a connection error (Section 5.4.1) of type FRAME_SIZE_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 only apply

to specific conditions and have no defined semantics in certain frame types.

The following error codes are defined:

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

PROTOCOL_ERROR (1): The endpoint detected an unspecific protocol error. This error is for use when a more specific error code is not available.

INTERNAL_ERROR (2): The endpoint encountered an unexpected internal error.

FLOW_CONTROL_ERROR (3): The endpoint detected that its peer violated the flow control protocol.

SETTINGS_TIMEOUT (4): The endpoint sent a SETTINGS frame, but did not receive a response in a timely manner. See Settings Synchronization (Section 6.5.3).

STREAM_CLOSED (5): The endpoint received a frame after a stream was half closed.

FRAME_SIZE_ERROR (6): The endpoint received a frame that was larger than the maximum size that it supports.

REFUSED_STREAM (7): The endpoint refuses the stream prior to performing any application processing, see Section 8.1.4 for details.

CANCEL (8): Used by the endpoint to indicate that the stream is no longer needed.

COMPRESSION_ERROR (9): The endpoint is unable to maintain the compression context for the connection.

CONNECT_ERROR (10): The connection established in response to a CONNECT request (Section 8.3) was reset or abnormally closed.

ENHANCE_YOUR_CALM (11): The endpoint detected that its peer is exhibiting a behavior over a given amount of time that has caused it to refuse to process further frames.

INADEQUATE_SECURITY (12): The underlying transport has properties that do not meet the minimum requirements imposed by this document (see Section 9.2) or the endpoint.

8. HTTP Message Exchanges

HTTP/2 is intended to be as compatible as possible with current uses of HTTP. This means that, from the perspective of the server and client applications, the features of the protocol are unchanged. To achieve this, all request and response semantics are preserved, although the syntax of conveying those semantics has changed.

Thus, the specification and requirements of HTTP/1.1 Semantics and Content [HTTP-p2], Conditional Requests [HTTP-p4], Range Requests [HTTP-p5], Caching [HTTP-p6] and Authentication [HTTP-p7] are applicable to HTTP/2. Selected portions of HTTP/1.1 Message Syntax and Routing [HTTP-p1], such as the HTTP and HTTPS URI schemes, are also applicable in HTTP/2, but the expression of those semantics for this protocol are defined in the sections below.

8.1. HTTP Request/Response Exchange

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 message headers; see [HTTP-p1], Section 3.2), and

2. zero or more DATA frames (containing the message payload; see [HTTP-p1], Section 3.3), and

3. optionally, one HEADERS frame, followed by zero or more CONTINUATION frames (containing the trailer-part, if present; see [HTTP-p1], Section 4.1.2).

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

Other frames (from any stream) MUST NOT occur between either HEADERS frame and the following CONTINUATION frames (if present), nor between CONTINUATION frames.

Otherwise, frames MAY be interspersed on the stream between these

frames, but those frames do not carry HTTP semantics. In particular, HEADERS frames (and any CONTINUATION frames that follow) other than the first and optional last frames in this sequence do not carry HTTP semantics.

Trailing header fields are carried in a header block that also terminates the stream. That is, a sequence starting with a HEADERS frame, followed by zero or more CONTINUATION frames, where the HEADERS frame bears an END_STREAM flag. Header blocks after the first that do not terminate the stream are not part of an HTTP request or response.

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 and ends with a frame bearing END_STREAM, which causes the stream to become "half closed" for the client. A response starts with a HEADERS frame and ends with a frame bearing END_STREAM, optionally followed by CONTINUATION frames, which places the stream in the "closed" state.

8.1.1. Informational Responses

The 1xx series of HTTP response status codes ([HTTP-p2], Section 6.2) are not supported in HTTP/2.

The most common use case for 1xx is using an Expect header field with a "100-continue" token (colloquially, "Expect/continue") to indicate that the client expects a 100 (Continue) non-final response status code, receipt of which indicates that the client should continue sending the request body if it has not already done so.

Typically, Expect/continue is used by clients wishing to avoid sending a large amount of data in a request body, only to have the request rejected by the origin server (thus leaving the connection potentially unusable).

HTTP/2 does not enable the Expect/continue mechanism; if the server sends a final status code to reject the request, it can do so without making the underlying connection unusable.

Note that this means HTTP/2 clients sending requests with bodies may waste at least one round trip of sent data when the request is rejected. This can be mitigated by restricting the amount of data sent for the first round trip by bandwidth-constrained clients, in anticipation of a final status code.

Other defined 1xx status codes are not applicable to HTTP/2. For example, the semantics of 101 (Switching Protocols) aren't suitable

to a multiplexed protocol. Likewise, 102 (Processing) is no longer necessary, because HTTP/2 has a separate means of keeping the connection alive.

This difference between protocol versions necessitates special handling by intermediaries that translate between them:

- An intermediary that gateways HTTP/1.1 to HTTP/2 MUST generate a 100 (Continue) response if a received request includes and Expect header field with a "100-continue" token ([HTTP-p2], Section 5.1.1), unless it can immediately generate a final status code. It MUST NOT forward the "100-continue" expectation in the request header fields.

- An intermediary that gateways HTTP/2 to HTTP/1.1 MAY add an Expect header field with a "100-continue" expectation when forwarding a request that has a body; see [HTTP-p2], Section 5.1.1 for specific requirements.

- An intermediary that gateways HTTP/2 to HTTP/1.1 MUST discard all other 1xx informational responses.

8.1.2. Examples

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

An HTTP GET request includes request header fields and no body 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 response header fields 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 request header fields and payload data is transmitted as one HEADERS frame, followed by zero or more CONTINUATION frames containing the request header fields, 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}                        content-type = image/jpeg

                                   CONTINUATION
                                     + END_HEADERS
                                       :authority = example.org
                                       :scheme = https
                                       content-length = 123

                                   DATA
                                     + END_STREAM
                                   {binary data}
```

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

A response that includes header fields and payload data 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}
```

Trailing header fields are sent as a header block after both the request or response header block and all the DATA frames have been sent. The HEADERS frame starting the trailers header block has the END_STREAM flag set.

```
  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
```

8.1.3. HTTP Header Fields

HTTP header fields carry information as a series of key-value pairs. For a listing of registered HTTP headers, see the Message Header Field Registry maintained at <http://www.iana.org/assignments/message-headers>.

While HTTP/1.x used the message start-line (see [HTTP-p1], Section 3.1) to convey the target URI and method of the request, and the status code for the response, HTTP/2 uses special pseudo-headers beginning with ":" for these tasks.

Just as in HTTP/1.x, header field names are strings of ASCII characters that are compared in a case-insensitive fashion. However, header field names MUST be converted to lowercase prior to their

encoding in HTTP/2. A request or response containing uppercase header field names MUST be treated as malformed (Section 8.1.3.5).

HTTP/2 does not use the Connection header field to indicate "hop-by-hop" header fields; in this protocol, connection-specific metadata is conveyed by other means. As such, a HTTP/2 message containing Connection MUST be treated as malformed (Section 8.1.3.5).

This means that an intermediary transforming an HTTP/1.x message to HTTP/2 will need to remove any header fields nominated by the Connection header field, along with the Connection header field itself. Such intermediaries SHOULD also remove other connection-specific header fields, such as Keep-Alive, Proxy-Connection, Transfer-Encoding and Upgrade, even if they are not nominated by Connection.

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

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.

8.1.3.1. Request Header Fields

HTTP/2 defines a number of header fields starting with a colon ':' character that carry information about the request target:

- The ":method" header field includes the HTTP method ([HTTP-p2], Section 4).

- The ":scheme" header field includes the scheme portion of the target URI ([RFC3986], Section 3.1).

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

  To ensure that the HTTP/1.1 request line can be reproduced accurately, this header field MUST be omitted when translating from an HTTP/1.1 request that has a request target in origin or asterisk form (see [HTTP-p1], Section 5.3). Clients that generate

  HTTP/2 requests directly SHOULD instead omit the "Host" header field. An intermediary that converts an HTTP/2 request to HTTP/1.1 MUST create a "Host" header field if one is not present in a request by copying the value of the ":authority" header field.

- The ":path" header field includes the path and query parts of the target URI (the "path-absolute" production from [RFC3986] and optionally a '?' character followed by the "query" production, see [RFC3986], Section 3.3 and [RFC3986], Section 3.4). This field MUST NOT be empty; URIs that do not contain a path component MUST include a value of '/', unless the request is an OPTIONS request in asterisk form, in which case the ":path" header field MUST include '*'.

All HTTP/2 requests MUST include exactly one valid value for the ":method", ":scheme", and ":path" header fields, unless this is a CONNECT request (Section 8.3). An HTTP request that omits mandatory header fields is malformed (Section 8.1.3.5).

Header field names that start with a colon are only valid in the HTTP/2 context. These are not HTTP header fields. Implementations MUST NOT generate header fields that start with a colon, but they MUST ignore any header field that starts with a colon. In particular, header fields with names starting with a colon MUST NOT be exposed as HTTP header fields.

HTTP/2 does not define a way to carry the version identifier that is included in the HTTP/1.1 request line.

8.1.3.2. Response Header Fields

A single ":status" header field is defined that carries the HTTP status code field (see [HTTP-p2], Section 6). This header field MUST be included in all responses, otherwise the response is malformed (Section 8.1.3.5).

HTTP/2 does not define a way to carry the version or reason phrase that is included in an HTTP/1.1 status line.

8.1.3.3. Header Field Ordering

HTTP Header Compression [COMPRESSION] does not preserve the order of header fields, because the relative order of header fields with different names is not important. However, the same header field can be repeated to form a list (see [HTTP-p1], Section 3.2.2), where the relative order of header field values is significant. This repetition can occur either as a single header field with a comma-

separated list of values, or as several header fields with a single value, or any combination thereof. Therefore, in the latter case, ordering needs to be preserved before compression takes place.

To preserve the order of multiple occurrences of a header field with the same name, its ordered values are concatenated into a single value using a zero-valued octet (0x0) to delimit them.

After decompression, header fields that have values containing zero octets (0x0) MUST be split into multiple header fields before being processed.

For example, the following HTTP/1.x header block:

```
Content-Type: text/html
Cache-Control: max-age=60, private
Cache-Control: must-revalidate
```

contains three Cache-Control directives; two in the first Cache-Control header field, and the last one in the second Cache-Control field. Before compression, they would need to be converted to a form similar to this (with 0x0 represented as "\0"):

```
cache-control: max-age=60, private\0must-revalidate
content-type: text/html
```

Note here that the ordering between Content-Type and Cache-Control is not preserved, but the relative ordering of the Cache-Control directives -- as well as the fact that the first two were comma-separated, while the last was on a different line -- is.

Header fields containing multiple values MUST be concatenated into a single value unless the ordering of that header field is known to be insignificant.

The special case of "set-cookie" - which does not form a comma-separated list, but can have multiple values - does not depend on ordering. The "set-cookie" header field MAY be encoded as multiple header field values, or as a single concatenated value.

8.1.3.4. Compressing the Cookie Header Field

The Cookie header field [COOKIE] can carry a significant amount of redundant data.

The Cookie header field uses a semi-colon (";") to delimit cookie-pairs (or "crumbs"). This header field doesn't follow the list construction rules in HTTP (see [HTTP-p1], Section 3.2.2), which

prevents cookie-pairs from being separated into different name-value pairs. This can significantly reduce compression efficiency as individual cookie-pairs are updated.

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 "; ").

The Cookie header field MAY be split using a zero octet (0x0), as defined in Section 8.1.3.3. When decoding, zero octets MUST be replaced with the cookie delimiter ("; ").

8.1.3.5. Malformed Messages

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

A request or response that includes an entity body can include a "content-length" header field. A request or response is also malformed if the value of a "content-length" header field does not equal the sum of the uncompressed DATA frame payload lengths that form the body.

Note: The "Content-Length" header field is set to the length of an entity body. Compression of DATA frames is a function of HTTP/2 that does not alter entities.

Intermediaries that process HTTP requests or responses (i.e., all intermediaries other than those acting as tunnels) MUST NOT forward a malformed request or response.

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

8.1.4. Request Reliability Mechanisms in HTTP/2

In HTTP/1.1, 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 enables a server to pre-emptively send (or "push") one or more associated responses to a client in response to a single request. This feature becomes particularly helpful when the server knows the client will need to have those responses available in order to fully process the response to the original request.

Pushing additional responses is optional, and is negotiated between individual endpoints. The SETTINGS_ENABLE_PUSH setting can be set to 0 to indicate that server push is disabled.

Because pushing responses is effectively hop-by-hop, an intermediary could receive pushed responses from the server and choose not to forward those on to the client. In other words, how to make use of the pushed responses is up to that intermediary. Equally, the

intermediary might choose to push additional responses 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. Clients MUST reject any attempt to change the SETTINGS_ENABLE_PUSH setting to a value other than "0" by treating the message as a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

A server can only push responses that are cacheable (see [HTTP-p6], Section 3); promised requests MUST be safe (see [HTTP-p2], Section 4.2.1) and MUST NOT include a request body.

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 header block that contains 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 a request body.

Pushed responses 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 8.1.3.1). The server MUST include a method in the ":method" 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" 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

promises before the DATA frames that contain the image links ensures that the client is able to see the promises before discovering embedded links. Similarly, if the server pushes responses referenced by the header block (for instance, in Link header fields), sending the push promises before sending the header block ensures that clients do not request them.

PUSH_PROMISE frames MUST NOT be sent by the client. PUSH_PROMISE frames can be sent by the server on any stream that was opened by the client. They MUST be sent on a stream that is in either the "open" or "half closed (remote)" state 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 header block.

8.2.2. Push Responses

After sending the PUSH_PROMISE frame, the server can begin delivering the pushed response as a response (Section 8.1.3.2) 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 an RST_STREAM frame, using either the CANCEL or REFUSED_STREAM codes, 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 the server is authorized to provide the response, see Section 10.1. For example, a server that offers a certificate for only the "example.com" DNS-ID or Common Name is not permitted to push a response for "https://www.example.org/doc".

8.3. The CONNECT Method

In HTTP/1.x, the pseudo-method CONNECT ([HTTP-p2], Section 4.3.6) 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. The HTTP header field mapping works as mostly as defined in Request Header Fields (Section 8.1.3.1), with a few differences. Specifically:

- The ":method" header field is set to "CONNECT".
- The ":scheme" and ":path" header fields MUST be omitted.
- The ":authority" header field contains the host and port to connect to (equivalent to the authority-form of the request-target of CONNECT requests, see [HTTP-p1], Section 5.3).

A proxy that supports CONNECT establishes a TCP connection [TCP] to the server identified in the ":authority" 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 [HTTP-p2], Section 4.3.6.

After the initial HEADERS frame sent by each peer, all subsequent DATA frames correspond to data sent on the TCP connection. The payload of any DATA frames sent by the client are 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 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 destination, where a destination is the IP address and port that 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), 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. [[anchor17: Need more text on how client certificates relate here, see issue #363.]]

Clients MAY use a single server connection to send requests for URIs with multiple different authority components as long as the server is authoritative (Section 10.1).

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-level 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.

9.2. Use of TLS Features

Implementations of HTTP/2 MUST support TLS 1.2 [TLS12]. 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. HTTP/2 clients MUST indicate the target

domain name when negotiating TLS.

The TLS implementation 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.

Implementations MUST negotiate - and therefore use - ephemeral cipher suites, such as ephemeral Diffie-Hellman (DHE) or the elliptic curve variant (ECDHE) with a minimum size of 2048 bits (DHE) or security level of 128 bits (ECDHE). Clients MUST accept DHE sizes of up to 4096 bits.

Implementations are encouraged not to negotiate TLS cipher suites with known vulnerabilities, such as [RC4].

An implementation that negotiates a TLS connection that does not meet the requirements in this section, or any policy-based constraints, SHOULD NOT negotiate HTTP/2. Removing HTTP/2 protocols from consideration could result in the removal of all protocols from the set of protocols offered by the client. This causes protocol negotiation failure, as described in Section 3.2 of [TLSALPN].

Due to implementation limitations, it might not be possible to fail TLS negotiation based on all of these requirements. An endpoint MUST terminate an HTTP/2 connection that is opened on a TLS session that does not meet these minimum requirements with a connection error (Section 5.4.1) of type INADEQUATE_SECURITY.

9.3. GZip Content-Encoding

Clients MUST support gzip compression for HTTP response bodies. Regardless of the value of the accept-encoding header field, a server MAY send responses with gzip encoding. A compressed response MUST still bear an appropriate content-encoding header field.

This effectively changes the implicit value of the Accept-Encoding header field ([HTTP-p2], Section 5.3.4) from "identity" to "identity,

gzip", however gzip encoding cannot be suppressed by including

";q=0". Intermediaries that perform translation from HTTP/2 to HTTP/1.1 MUST decompress payloads unless the request includes an Accept-Encoding value that includes "gzip".

10. Security Considerations

10.1. Server Authority

A client is only able to accept HTTP/2 responses from servers that are authoritative for those resources. This is particularly important for server push (Section 8.2), where the client validates the PUSH_PROMISE before accepting the response.

HTTP/2 relies on the HTTP/1.1 definition of authority for determining whether a server is authoritative in providing a given response, see [HTTP-p1], Section 9.1. This relies on local name resolution for the "http" URI scheme, and the offered server identity for the "https" scheme (see [RFC2818], Section 3).

A client MUST NOT use, in any way, resources provided by a server that is not authoritative for those resources.

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 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. 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 which 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 3.5) contains a string that is designed to confuse HTTP/1.1 servers, but no special protection is offered for other protocols. A server that is willing to ignore parts of an HTTP/1.1 request containing an Upgrade header field could be exposed to a cross-protocol attack.

10.3. Intermediary Encapsulation Attacks

HTTP/2 header field names and values are encoded as sequences of octets with a length prefix. This enables HTTP/2 to carry any string of octets as the name or value of a header field. An intermediary that translates HTTP/2 requests or responses into HTTP/1.1 directly could permit the creation of corrupted HTTP/1.1 messages. An attacker might exploit this behavior to cause the intermediary to create HTTP/1.1 messages with illegal header fields, extra header fields, or even new messages that are entirely falsified.

Header field names or values that contain characters not permitted by HTTP/1.1, including carriage return (U+000D) or line feed (U+000A) MUST NOT be translated verbatim by an intermediary, as stipulated in [HTTP-p1], Section 3.2.4.

Translation from HTTP/1.x to HTTP/2 does not produce the same opportunity to an attacker. Intermediaries that perform translation to HTTP/2 MUST remove any instances of the "obs-fold" production from header field values.

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) are never cached or used.

10.5. Denial of Service Considerations

An HTTP/2 connection can demand a greater commitment of resources to operate than a HTTP/1.1 connection. The use of header 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. Processing capacity cannot be guarded in the same fashion.

The SETTINGS frame can be abused to cause a peer to expend additional processing time. This might be done by pointlessly changing SETTINGS parameters, setting multiple undefined parameters, or changing the same setting multiple times in the same frame. WINDOW_UPDATE, PRIORITY, or BLOCKED frames can be abused to cause an unnecessary waste of resources. A server might erroneously issue ALTSVC frames for origins on which it cannot be authoritative to generate excess work for clients.

Large numbers of small or empty frames can be abused to cause a peer to expend time processing frame headers. Note however that some uses are entirely legitimate, such as the sending of an empty DATA frame to end a stream.

Header compression also offers some opportunities to waste processing resources; see [COMPRESSION] for more details on potential abuses.

Limits in SETTINGS parameters 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.

All these features - i.e., SETTINGS changes, small frames, header compression - have legitimate uses. These features become a burden only when they are used unnecessarily or to excess.

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

10.6. Use of Compression

HTTP/2 enables greater use of compression for both header fields

(Section 4.3) and response bodies (Section 9.3). Compression can allow an attacker to recover secret data when it is compressed in the same context as data under attacker control.

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.

Intermediaries MUST NOT alter the compression of DATA frames unless additional information is available that allows the intermediary to identify the source of data. In particular, frames that are not compressed cannot be compressed, and frames that are separately compressed cannot be merged into a single frame. Compressed frames MAY be decompressed or split into multiple frames.

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 might be provided by TLS [TLS12]. 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 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 (see for example, [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; or padding payloads to a fixed size exposes information as payload sizes cross the fixed size boundary, which could be possible if an attacker can control plaintext.

Intermediaries SHOULD NOT remove padding, though an intermediary MAY remove padding and add differing amounts if the intent is to improve the protections padding affords.

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. This includes the value of settings, the manner in which flow control windows are managed, the way priorities are allocated to streams, timing of reactions to stimulus, and handling of any optional features.

As far as this creates observable differences in behavior, they could be used as a basis for fingerprinting a specific client, as defined in <http://www.w3.org/TR/html5/introduction.html#fingerprint>.

11. IANA Considerations

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

This document establishes a registry for error codes. This new registry is entered into a new "Hypertext Transfer Protocol (HTTP) 2 Parameters" 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 3.5).

11.1. Registration of HTTP/2 Identification Strings

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

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

Protocol: HTTP/2 over TLS

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

Specification: This document (RFCXXXX)

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 (RFCXXXX)

11.2. 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 "Expert Review" policy [RFC5226].

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:

Error Code: The 32-bit error code value.

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

Description: A description of the conditions where the error code is applicable.

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

An initial set of error code registrations can be found in Section 7.

11.3. HTTP2-Settings Header Field Registration

This section registers the "HTTP2-Settings" header field in the Permanent Message Header Field Registry [BCP90].

Header field name: HTTP2-Settings

Applicable protocol: http

Status: standard

Author/Change controller: IETF

Specification document(s): Section 3.2.1 of this document

Related information: This header field is only used by an HTTP/2 client for Upgrade-based negotiation.

11.4. PRI Method Registration

This section registers the "PRI" method in the HTTP Method Registry [HTTP-p2].

Method Name: PRI

Safe No

Idempotent No

Specification document(s) Section 3.5 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.

12. 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, Jonathan Leighton (SPDY contributors).

- Gabriel Montenegro and Willy Tarreau (Upgrade mechanism).

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

- Mark Nottingham, Julian Reschke, James Snell, Jeff Pinner, Mike Bishop, Herve Ruellan (Substantial editorial contributions).

- Alexey Melnikov was an editor of this document during 2013.

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

13. References

13.1. Normative References

[ALT-SVC] Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", draft-ietf-httpbis-alt-svc-01

(work in progress), April 2014.

[COMPRESSION] Ruellan, H. and R. Peon, "HPACK - Header Compression for HTTP/2", draft-ietf-httpbis-header-compression-07 (work in progress), April 2014.

[COOKIE] Barth, A., "HTTP State Management Mechanism", RFC 6265, April 2011.

[GZIP] Deutsch, P., Gailly, J-L., Adler, M., Deutsch, L., and G. Randers-Pehrson, "GZIP file format specification version 4.3", RFC 1952, May 1996.

[HTTP-p1] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", draft-ietf-httpbis-p1-messaging-26 (work in progress), February 2014.

[HTTP-p2] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", draft-ietf-httpbis-p2-semantics-26 (work in progress), February 2014.

[HTTP-p4] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests", draft-ietf-httpbis-p4-conditional-26 (work in progress), February 2014.

[HTTP-p5] Fielding, R., Ed., Lafon, Y., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Range Requests", draft-ietf-httpbis-p5-range-26 (work in progress), February 2014.

[HTTP-p6] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Caching", draft-ietf-httpbis-p6-cache-26 (work in progress), February 2014.

[HTTP-p7] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Authentication", draft-ietf-httpbis-p7-auth-26 (work in progress), February 2014.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate

Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.

[RFC5234] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008.

[RFC6454] Barth, A., "The Web Origin Concept", RFC 6454, December 2011.

[TCP] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

[TLS-EXT] Eastlake, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, January 2011.

[TLS12] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

[TLSALPN] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application Layer Protocol Negotiation Extension", draft-ietf-tls-applayerprotoneg-05 (work in progress), March 2014.

[UTF-8] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.

13.2. Informative References

[BCP90] Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, September 2004.

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

[IDNA] Klensin, J., "Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework", RFC 5890, August 2010.

[RC4] Rivest, R., "The RC4 encryption algorithm", RSA Data Security, Inc. , March 1992.

[RFC1323] Jacobson, V., Braden, B., and D. Borman, "TCP Extensions for High Performance", RFC 1323, May 1992.

[RFC3749] Hollenbeck, S., "Transport Layer Security Protocol Compression Methods", RFC 3749, May 2004.

[TALKING] Huang, L-S., 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 TLS and DTLS", draft-sheffer-tls-bcp-02 (work in progress), February 2014.

Appendix A. Change Log (to be removed by RFC Editor before publication)

A.1. Since draft-ietf-httpbis-http2-11

Added BLOCKED frame (at risk).

Simplified priority scheme.

Added DATA per-frame GZip compression.

A.2. Since draft-ietf-httpbis-http2-10

Changed "connection header" to "connection preface" to avoid confusion.

Added dependency-based stream prioritization.

Added "h2c" identifier to distinguish between cleartext and secured HTTP/2.

Adding missing padding to PUSH_PROMISE.

Integrate ALTSVC frame and supporting text.

Dropping requirement on "deflate" Content-Encoding.

Improving security considerations around use of compression.

A.3. Since draft-ietf-httpbis-http2-09

Adding padding for data frames.

Renumbering frame types, error codes, and settings.

Adding INADEQUATE_SECURITY error code.

Updating TLS usage requirements to 1.2; forbidding TLS compression.

Removing extensibility for frames and settings.

Changing setting identifier size.

Removing the ability to disable flow control.

Changing the protocol identification token to "h2".

Changing the use of :authority to make it optional and to allow userinfo in non-HTTP cases.

Allowing split on 0x0 for Cookie.

Reserved PRI method in HTTP/1.1 to avoid possible future collisions.

A.4. Since draft-ietf-httpbis-http2-08

Added cookie crumbling for more efficient header compression.

Added header field ordering with the value-concatenation mechanism.

A.5. Since draft-ietf-httpbis-http2-07

Marked draft for implementation.

A.6. Since draft-ietf-httpbis-http2-06

Adding definition for CONNECT method.

Constraining the use of push to safe, cacheable methods with no request body.

Changing from :host to :authority to remove any potential confusion.

Adding setting for header compression table size.

Adding settings acknowledgement.

Removing unnecessary and potentially problematic flags from CONTINUATION.

Added denial of service considerations.

A.7. Since draft-ietf-httpbis-http2-05

Marking the draft ready for implementation.

Renumbering END_PUSH_PROMISE flag.

Editorial clarifications and changes.

A.8. Since draft-ietf-httpbis-http2-04

Added CONTINUATION frame for HEADERS and PUSH_PROMISE.

PUSH_PROMISE is no longer implicitly prohibited if SETTINGS_MAX_CONCURRENT_STREAMS is zero.

Push expanded to allow all safe methods without a request body.

Clarified the use of HTTP header fields in requests and responses. Prohibited HTTP/1.1 hop-by-hop header fields.

Requiring that intermediaries not forward requests with missing or illegal routing :-headers.

Clarified requirements around handling different frames after stream close, stream reset and GOAWAY.

Added more specific prohibitions for sending of different frame types in various stream states.

Making the last received setting value the effective value.

Clarified requirements on TLS version, extension and ciphers.

A.9. Since draft-ietf-httpbis-http2-03

Committed major restructuring atrocities.

Added reference to first header compression draft.

Added more formal description of frame lifecycle.

Moved END_STREAM (renamed from FINAL) back to HEADERS/DATA.

Removed HEADERS+PRIORITY, added optional priority to HEADERS frame.

Added PRIORITY frame.

A.10. Since draft-ietf-httpbis-http2-02

Added continuations to frames carrying header blocks.

Replaced use of "session" with "connection" to avoid confusion with other HTTP stateful concepts, like cookies.

Removed "message".

Switched to TLS ALPN from NPN.

Editorial changes.

A.11. Since draft-ietf-httpbis-http2-01

Added IANA considerations section for frame types, error codes and settings.

Removed data frame compression.

Added PUSH_PROMISE.

Added globally applicable flags to framing.

Removed zlib-based header compression mechanism.

Updated references.

Clarified stream identifier reuse.

Removed CREDENTIALS frame and associated mechanisms.

Added advice against naive implementation of flow control.

Added session header section.

Restructured frame header. Removed distinction between data and control frames.

Altered flow control properties to include session-level limits.

Added note on cacheability of pushed resources and multiple tenant servers.

Changed protocol label form based on discussions.

A.12. Since draft-ietf-httpbis-http2-00

Changed title throughout.

Removed section on Incompatibilities with SPDY draft#2.

Changed INTERNAL_ERROR on GOAWAY to have a value of 2 <https://groups.google.com/forum/?fromgroups#!topic/spdy-dev/cfUef2gL3iU>.

Replaced abstract and introduction.

Added section on starting HTTP/2.0, including upgrade mechanism.

Removed unused references.

Added flow control principles (Section 5.2.1) based on <http://tools.ietf.org/html/draft-montenegro-httpbis-http2-fc-principles-01>.

A.13. Since draft-mbelshe-httpbis-spdy-00

Adopted as base for draft-ietf-httpbis-http2.

Updated authors/editors list.

Added status note.

Authors' Addresses

Mike Belshe
Twist

EMail: mbelshe@chromium.org

Roberto Peon
Google, Inc

EMail: fenix@google.com

Martin Thomson (editor)
Mozilla
Suite 300
650 Castro Street
Mountain View, CA 94041

US

EMail: martin.thomson@gmail.com