

HTTPbis Working Group  
Internet-Draft  
Intended status: Standards Track  
Expires: December 19, 2014

M. Belshe  
Twist  
R. Peon  
Google, Inc  
M. Thomson, Ed.  
Mozilla  
June 17, 2014

## **Hypertext Transfer Protocol version 2 draft-ietf-httpbis-http2-13**

### 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 specification 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](mailto:ietf-http-wg@w3.org)), which is archived at [\[1\]](#).

Working Group information can be found at [\[2\]](#); that specific to HTTP/2 are at [\[3\]](#).

The changes in this draft are summarized in [Appendix A](#).

### 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 December 19, 2014.

## Copyright Notice

Copyright (c) 2014 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to [BCP 78](#) and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

## Table of Contents

<a href="#">1.</a>	<a href="#">Introduction</a>	<a href="#">4</a>
<a href="#">2.</a>	<a href="#">HTTP/2 Protocol Overview</a>	<a href="#">5</a>
<a href="#">2.1.</a>	<a href="#">Document Organization</a>	<a href="#">6</a>
<a href="#">2.2.</a>	<a href="#">Conventions and Terminology</a>	<a href="#">6</a>
<a href="#">3.</a>	<a href="#">Starting HTTP/2</a>	<a href="#">7</a>
<a href="#">3.1.</a>	<a href="#">HTTP/2 Version Identification</a>	<a href="#">8</a>
<a href="#">3.2.</a>	<a href="#">Starting HTTP/2 for "http" URIs</a>	<a href="#">9</a>
<a href="#">3.2.1.</a>	<a href="#">HTTP2-Settings Header Field</a>	<a href="#">10</a>
<a href="#">3.3.</a>	<a href="#">Starting HTTP/2 for "https" URIs</a>	<a href="#">11</a>
<a href="#">3.4.</a>	<a href="#">Starting HTTP/2 with Prior Knowledge</a>	<a href="#">11</a>
<a href="#">3.5.</a>	<a href="#">HTTP/2 Connection Preface</a>	<a href="#">11</a>
<a href="#">4.</a>	<a href="#">HTTP Frames</a>	<a href="#">12</a>
<a href="#">4.1.</a>	<a href="#">Frame Format</a>	<a href="#">12</a>
<a href="#">4.2.</a>	<a href="#">Frame Size</a>	<a href="#">13</a>
<a href="#">4.3.</a>	<a href="#">Header Compression and Decompression</a>	<a href="#">14</a>
<a href="#">5.</a>	<a href="#">Streams and Multiplexing</a>	<a href="#">15</a>
<a href="#">5.1.</a>	<a href="#">Stream States</a>	<a href="#">15</a>
<a href="#">5.1.1.</a>	<a href="#">Stream Identifiers</a>	<a href="#">20</a>
<a href="#">5.1.2.</a>	<a href="#">Stream Concurrency</a>	<a href="#">21</a>
<a href="#">5.2.</a>	<a href="#">Flow Control</a>	<a href="#">21</a>
<a href="#">5.2.1.</a>	<a href="#">Flow Control Principles</a>	<a href="#">21</a>
<a href="#">5.2.2.</a>	<a href="#">Appropriate Use of Flow Control</a>	<a href="#">22</a>
<a href="#">5.3.</a>	<a href="#">Stream priority</a>	<a href="#">23</a>
<a href="#">5.3.1.</a>	<a href="#">Stream Dependencies</a>	<a href="#">24</a>
<a href="#">5.3.2.</a>	<a href="#">Dependency Weighting</a>	<a href="#">25</a>
<a href="#">5.3.3.</a>	<a href="#">Reprioritization</a>	<a href="#">25</a>
<a href="#">5.3.4.</a>	<a href="#">Prioritization State Management</a>	<a href="#">26</a>
<a href="#">5.3.5.</a>	<a href="#">Default Priorities</a>	<a href="#">27</a>
<a href="#">5.4.</a>	<a href="#">Error Handling</a>	<a href="#">27</a>



5.4.1.	Connection Error Handling . . . . .	27
5.4.2.	Stream Error Handling . . . . .	28
5.4.3.	Connection Termination . . . . .	28
5.5.	Extending HTTP/2 . . . . .	28
6.	Frame Definitions . . . . .	29
6.1.	DATA . . . . .	29
6.2.	HEADERS . . . . .	31
6.3.	PRIORITY . . . . .	33
6.4.	RST_STREAM . . . . .	34
6.5.	SETTINGS . . . . .	35
6.5.1.	SETTINGS Format . . . . .	36
6.5.2.	Defined SETTINGS Parameters . . . . .	36
6.5.3.	Settings Synchronization . . . . .	37
6.6.	PUSH_PROMISE . . . . .	38
6.7.	PING . . . . .	40
6.8.	GOAWAY . . . . .	41
6.9.	WINDOW_UPDATE . . . . .	43
6.9.1.	The Flow Control Window . . . . .	44
6.9.2.	Initial Flow Control Window Size . . . . .	45
6.9.3.	Reducing the Stream Window Size . . . . .	46
6.10.	CONTINUATION . . . . .	47
7.	Error Codes . . . . .	47
8.	HTTP Message Exchanges . . . . .	49
8.1.	HTTP Request/Response Exchange . . . . .	49
8.1.1.	Informational Responses . . . . .	50
8.1.2.	HTTP Header Fields . . . . .	51
8.1.3.	Examples . . . . .	55
8.1.4.	Request Reliability Mechanisms in HTTP/2 . . . . .	58
8.2.	Server Push . . . . .	59
8.2.1.	Push Requests . . . . .	59
8.2.2.	Push Responses . . . . .	60
8.3.	The CONNECT Method . . . . .	61
9.	Additional HTTP Requirements/Considerations . . . . .	62
9.1.	Connection Management . . . . .	62
9.1.1.	Connection Reuse . . . . .	63
9.1.2.	The 421 (Not Authoritative) Status Code . . . . .	64
9.2.	Use of TLS Features . . . . .	64
9.2.1.	TLS Features . . . . .	64
9.2.2.	TLS Cipher Suites . . . . .	65
10.	Security Considerations . . . . .	65
10.1.	Server Authority . . . . .	65
10.2.	Cross-Protocol Attacks . . . . .	66
10.3.	Intermediary Encapsulation Attacks . . . . .	66
10.4.	Cacheability of Pushed Responses . . . . .	67
10.5.	Denial of Service Considerations . . . . .	67
10.5.1.	Limits on Header Block Size . . . . .	68
10.6.	Use of Compression . . . . .	69
10.7.	Use of Padding . . . . .	69



<a href="#">10.8.</a>	Privacy Considerations . . . . .	<a href="#">70</a>
<a href="#">11.</a>	IANA Considerations . . . . .	<a href="#">70</a>
<a href="#">11.1.</a>	Registration of HTTP/2 Identification Strings . . . . .	<a href="#">70</a>
<a href="#">11.2.</a>	Frame Type Registry . . . . .	<a href="#">71</a>
<a href="#">11.3.</a>	Settings Registry . . . . .	<a href="#">72</a>
<a href="#">11.4.</a>	Error Code Registry . . . . .	<a href="#">72</a>
<a href="#">11.5.</a>	HTTP2-Settings Header Field Registration . . . . .	<a href="#">73</a>
<a href="#">11.6.</a>	PRI Method Registration . . . . .	<a href="#">74</a>
<a href="#">11.7.</a>	The 421 Not Authoritative HTTP Status Code . . . . .	<a href="#">74</a>
<a href="#">12.</a>	Acknowledgements . . . . .	<a href="#">74</a>
<a href="#">13.</a>	References . . . . .	<a href="#">75</a>
<a href="#">13.1.</a>	Normative References . . . . .	<a href="#">75</a>
<a href="#">13.2.</a>	Informative References . . . . .	<a href="#">76</a>
<a href="#">13.3.</a>	URIs . . . . .	<a href="#">77</a>
<a href="#">Appendix A.</a>	Change Log (to be removed by RFC Editor before publication) . . . . .	<a href="#">78</a>
<a href="#">A.1.</a>	Since <a href="#">draft-ietf-httpbis-http2-12</a> . . . . .	<a href="#">78</a>
<a href="#">A.2.</a>	Since <a href="#">draft-ietf-httpbis-http2-11</a> . . . . .	<a href="#">78</a>
<a href="#">A.3.</a>	Since <a href="#">draft-ietf-httpbis-http2-10</a> . . . . .	<a href="#">78</a>
<a href="#">A.4.</a>	Since <a href="#">draft-ietf-httpbis-http2-09</a> . . . . .	<a href="#">79</a>
<a href="#">A.5.</a>	Since <a href="#">draft-ietf-httpbis-http2-08</a> . . . . .	<a href="#">79</a>
<a href="#">A.6.</a>	Since <a href="#">draft-ietf-httpbis-http2-07</a> . . . . .	<a href="#">79</a>
<a href="#">A.7.</a>	Since <a href="#">draft-ietf-httpbis-http2-06</a> . . . . .	<a href="#">79</a>
<a href="#">A.8.</a>	Since <a href="#">draft-ietf-httpbis-http2-05</a> . . . . .	<a href="#">80</a>
<a href="#">A.9.</a>	Since <a href="#">draft-ietf-httpbis-http2-04</a> . . . . .	<a href="#">80</a>
<a href="#">A.10.</a>	Since <a href="#">draft-ietf-httpbis-http2-03</a> . . . . .	<a href="#">80</a>
<a href="#">A.11.</a>	Since <a href="#">draft-ietf-httpbis-http2-02</a> . . . . .	<a href="#">81</a>
<a href="#">A.12.</a>	Since <a href="#">draft-ietf-httpbis-http2-01</a> . . . . .	<a href="#">81</a>
<a href="#">A.13.</a>	Since <a href="#">draft-ietf-httpbis-http2-00</a> . . . . .	<a href="#">82</a>
<a href="#">A.14.</a>	Since <a href="#">draft-mbelshe-httpbis-spdy-00</a> . . . . .	<a href="#">82</a>

## 1. Introduction

The Hypertext Transfer Protocol (HTTP) is a wildly successful protocol. However, the HTTP/1.1 message format ([\[RFC7230\]](#), [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 [[TCP](#)] congestion window to quickly fill. This can result in excessive latency when multiple requests are made on a single new TCP connection.

This specification 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 efficient 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 type serves a different purpose. For example, HEADERS and DATA frames form the basis of HTTP requests and responses ([Section 8.1](#)); other frame types like SETTINGS, WINDOW\_UPDATE, and PUSH\_PROMISE are used in support of other HTTP/2 features.

Multiplexing of requests is achieved by having each HTTP request-response 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.

## **[2.1.](#) Document Organization**

The HTTP/2 specification is split into four parts:

- o Starting HTTP/2 ([Section 3](#)) covers how an HTTP/2 connection is initiated.
- o The framing ([Section 4](#)) and streams ([Section 5](#)) layers describe the way HTTP/2 frames are structured and formed into multiplexed streams.
- o Frame ([Section 6](#)) and error ([Section 7](#)) definitions include details of the frame and error types used in HTTP/2.
- o 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.

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.

Finally, the terms "gateway", "intermediary", "proxy", and "tunnel" are defined in [Section 2.3 of \[RFC7230\]](#).

### **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.

- o The string "h2" identifies the protocol where HTTP/2 uses TLS [[TLS12](#)]. This identifier is used in the TLS application layer protocol negotiation extension (ALPN) [[TLSALPN](#)] field and any place that HTTP/2 over TLS is identified.

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

- o 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.

[[CREF1: 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 \[RFC7230\]](#). Experimenters are encouraged to coordinate their experiments on the [ietf-http-wg@w3.org](mailto: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 \[RFC7230\]](#)). 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 / 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 MUST ignore a "h2" token in an Upgrade header field. Presence of a token with "h2" implies HTTP/2 over TLS, which is instead negotiated as described in [Section 3.3](#).

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.

```
HTTP2-Settings    = token68
```

A server MUST reject an attempt to upgrade if this header field is not present. A server MUST NOT send this header field.

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 \[RFC7235\]](#).

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].

HTTP/2 over TLS uses the "h2" application token. The "h2c" token MUST NOT be sent by a client or selected by a server.

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.

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 establishment of HTTP/2 connections over cleartext TCP; implementations that support HTTP/2 over TLS MUST use protocol negotiation in TLS [TLSALPN].

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.

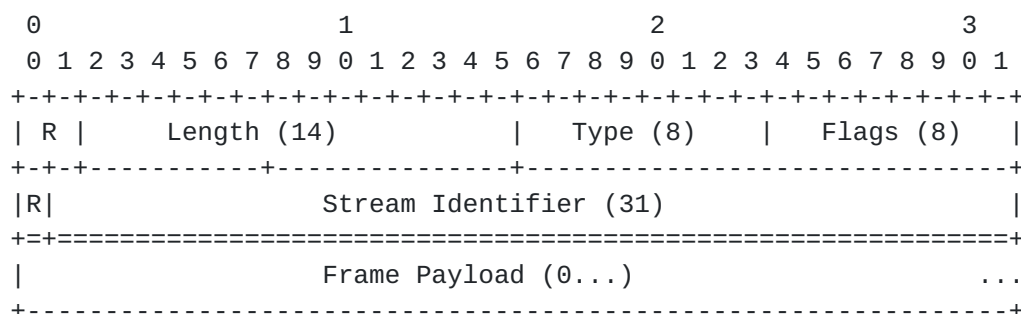
Clients and servers MUST terminate the TCP connection if either peer does not begin with a valid connection preface. A GOAWAY frame ([Section 6.8](#)) can be omitted if it is clear that the peer is not using HTTP/2.

## 4. HTTP Frames

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

#### 4.1. Frame Format

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



## Frame Layout



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

Flags: An 8-bit field reserved for 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 MUST be capable of receiving and minimally processing frames up to this maximum size.

Certain frame types, such as PING ([Section 6.7](#)), impose additional limits on the amount of payload data allowed.

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 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 8.1.2.3](#)), this preserves the relative order of values for that header field.

The Cookie header field [[COOKIE](#)] is treated specially by the HTTP mapping (see [Section 8.1.2.4](#)).

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:

- o a single HEADERS or PUSH\_PROMISE frame, with the END\_HEADERS flag set, or
- o 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. This allows a header block to be logically equivalent to a single frame.



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

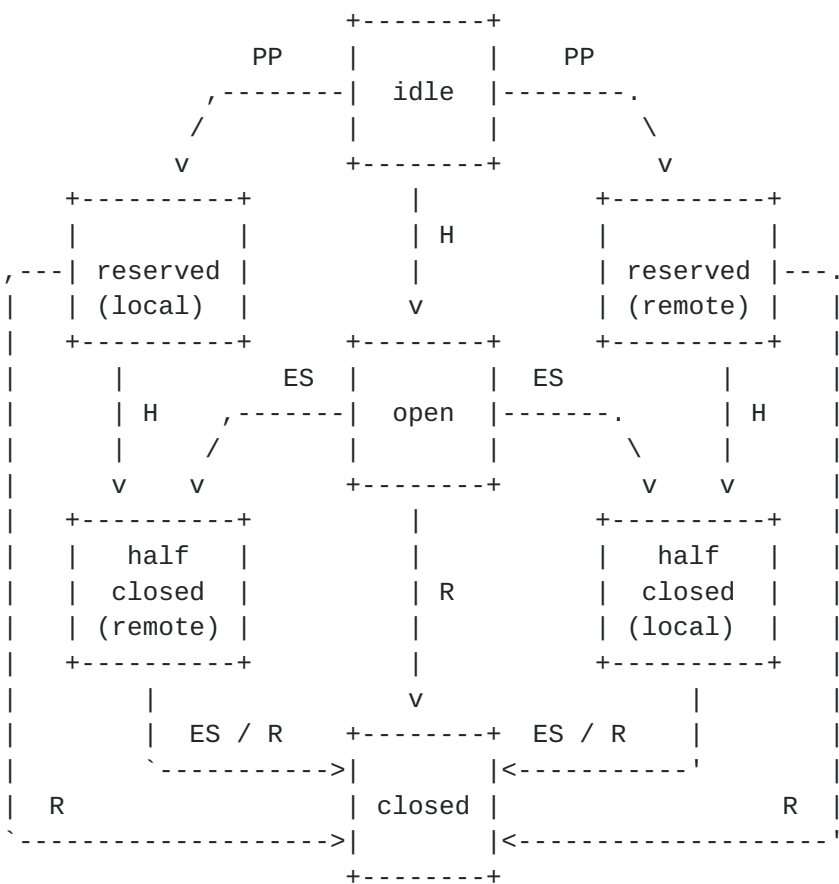
A "stream" is an independent, bi-directional sequence of frames exchanged between the client and server within an HTTP/2 connection. Streams have several important characteristics:

- o A single HTTP/2 connection can contain multiple concurrently open streams, with either endpoint interleaving frames from multiple streams.
- o Streams can be established and used unilaterally or shared by either the client or server.
- o Streams can be closed by either endpoint.
- o The order in which frames are sent on a stream is significant. Recipients process frames in the order they are received. In particular, the order of HEADERS, and DATA frames is semantically significant.
- o 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

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

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)".

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. Only WINDOW\_UPDATE, PRIORITY and RST\_STREAM frames can be sent in this state.

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 receiver can ignore WINDOW\_UPDATE frames in this state, which might arrive for a short period after a frame bearing the END\_STREAM flag is sent.

PRIORITY frames received in this state are used to reprioritize streams that depend on the current stream.

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 WINDOW\_UPDATE, PRIORITY or RST\_STREAM, 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 other than PRIORITY 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 frame with the END\_STREAM flag set MUST treat that as a connection error ([Section 5.4.1](#)) of type STREAM\_CLOSED, unless the frame is permitted as described below.

WINDOW\_UPDATE 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 frames of these types. Endpoints MUST ignore WINDOW\_UPDATE 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.

PRIORITY frames can be sent on closed streams to prioritize streams that are dependent on the closed stream. Endpoints SHOULD process PRIORITY frame, though they can be ignored if the stream has been removed from the dependency tree (see [Section 5.3.4](#)).

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 stream.

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 cannot 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 an endpoint exhausting the available range of stream identifiers. A client that is unable to establish a new stream identifier can establish a new connection for new streams. A server that is unable to establish a new stream identifier can send a GOAWAY frame so that the client is forced to open a new connection for new streams.



### **5.1.2. Stream Concurrency**

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

Streams that are in the "open" state, or either of the "half closed" states count toward the maximum number of streams that an endpoint is permitted to open. Streams in any of these three states count toward the limit advertised in the `SETTINGS_MAX_CONCURRENT_STREAMS` setting. Streams in either of the "reserved" states do not count toward the stream limit.

Endpoints **MUST NOT** exceed the limit set by their peer. An endpoint that receives a `HEADERS` frame that causes their advertised concurrent stream limit to be exceeded **MUST** treat this as a stream error ([Section 5.4.2](#)). An endpoint that wishes to reduce the value of `SETTINGS_MAX_CONCURRENT_STREAMS` to a value that is below the current number of open streams can either close streams that exceed the new value or allow streams to complete.

## **5.2. Flow Control**

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

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

### **5.2.1. Flow Control Principles**

HTTP/2 stream flow control aims to allow 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 defines only the format and semantics of the WINDOW\_UPDATE frame ([Section 6.9](#)). This document does not stipulate how a receiver decides when to send this frame or the value that it sends. Nor does it specify how a sender chooses to send packets. Implementations are able to select any algorithm that suits their needs.

Implementations are also responsible for 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. This effectively disables flow control for that receiver. Conversely, a sender is always subject to the flow control window advertised by the receiver.



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

### 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.

[[CREF2: Note that stream dependencies have not yet been validated in practice. The theory might be fairly sound, but there are no implementations currently sending these. If it turns out that they are not useful, or actively harmful, implementations will be requested to avoid creating stream dependencies.]]

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 is given a stream dependency of 0x0. In other words, the non-existent stream 0 forms the root of the tree.

A stream that depends on another stream is a dependent stream. The stream upon which a stream is dependent is a parent stream.

When assigning a dependency on another stream, the stream is added as a new dependency of the parent stream. Dependent streams that share the same parent are not order with respect to each other. 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 in any order.



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 its parent stream, causing other dependencies to become dependent on the prioritized stream. In the previous example, if stream D is created with an exclusive dependency on stream A, this results in D becoming the dependency parent of B and C.



Example of Exclusive Dependency Creation

Inside the dependency tree, a dependent stream SHOULD only be allocated resources if all of the streams that it depends on (the chain of parent streams up to 0x0) are either closed, or it is not possible to make progress on them.

A stream cannot depend on itself. An endpoint MUST treat this as a stream error ([Section 5.4.2](#)) of type `PROTOCOL_ERROR`.



### 5.3.2. Dependency Weighting

All dependent streams are allocated an integer weight between 1 to 256 (inclusive).

Streams with the same parent SHOULD be allocated resources proportionally based on their weight. Thus, if stream B depends on stream A with weight 4, 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.

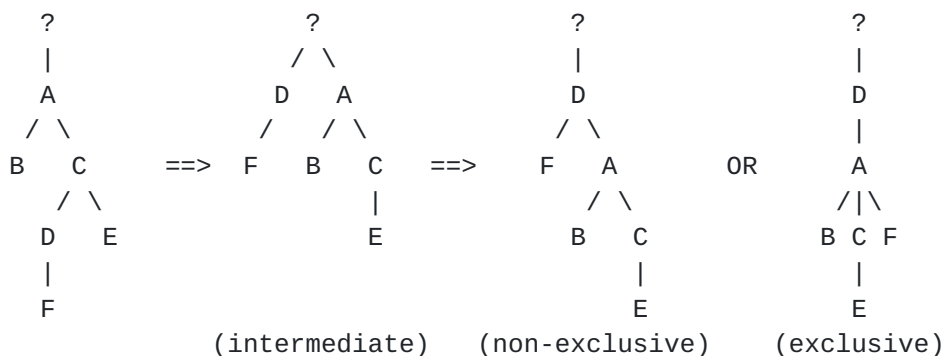
### 5.3.3. Reprioritization

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

Dependent streams move with their parent stream if the parent is reprioritized. Setting a dependency with the exclusive flag for a reprioritized stream moves all the dependencies of the new parent stream to become dependent on the reprioritized stream.

If a stream is made dependent on one of its own dependencies, the formerly dependent stream is first moved to be dependent on the reprioritized stream's previous parent. The moved dependency retains its weight.

For example, consider 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. All other dependency relationships stay the same, except for F, which becomes dependent on A if the reprioritization is exclusive.



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 parent of the closed stream. The weights of new dependencies are recalculated 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 with the same parent 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 parent, and streams C and D both depend on stream A. Prior to the removal of stream A, if streams A and D are unable to proceed, then stream C receives all the resources dedicated to stream A. If stream A is removed from the tree, the weight of stream A is divided between streams C and D. If stream D is still unable to proceed, this results in stream C receiving a reduced proportion of resources. For equal starting weights, C receives one third, rather than one half, of available resources.

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 had any associated priority information destroyed, then the dependent stream is instead assigned a default priority. This potentially creates suboptimal prioritization, since the stream could be given a priority that is higher than intended.

To avoid these problems, 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 default 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:

- o An error condition that renders the entire connection unusable is a connection error.
- o 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, providing that circumstances permit it.



#### **5.4.2. Stream Error Handling**

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

An endpoint that detects a stream error sends a RST\_STREAM frame ([Section 6.4](#)) that contains the stream identifier of the stream where the error occurred. The RST\_STREAM frame includes an error code that indicates the type of error.

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

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

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.

### **5.5. Extending HTTP/2**

HTTP/2 permits extension of the protocol. Protocol extensions can be used to provide additional services or alter any aspect of the protocol, within the limitations described in this section. Extensions are effective only within the scope of a single HTTP/2 connection.

Extensions are permitted to use new frame types ([Section 4.1](#)), new settings ([Section 6.5.2](#)), new error codes ([Section 7](#)), or new header fields that start with a colon (:). Of these, registries are established for frame types ([Section 11.2](#)), settings ([Section 11.3](#)) and error codes ([Section 11.4](#)).

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



that have unknown or unsupported types. This means that any of these extension points can be safely used by extensions without prior arrangement or negotiation.

However, extensions that could change the semantics of existing protocol components **MUST** be negotiated before being used. For example, an extension that changes the layout of the HEADERS frame cannot be used until the peer has given a positive signal that this is acceptable. In this case, it could also be necessary to coordinate when the revised layout comes into effect. Note that treating any frame other than DATA frames as flow controlled is such a change in semantics, and can only be done through negotiation.

This document doesn't mandate a specific method for negotiating the use of an extension, but notes that a setting ([Section 6.5.2](#)) could be used for that purpose. If both peers set a value that indicates willingness to use the extension, then the extension can be used. If a setting is used for extension negotiation, the initial value **MUST** be defined so that the extension is initially disabled.

## **[6.](#) Frame Definitions**

This specification defines a number of frame types, each identified by a unique 8-bit type code. Each frame type serves a distinct purpose 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 obscure the size of messages.













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`.

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

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](#)).

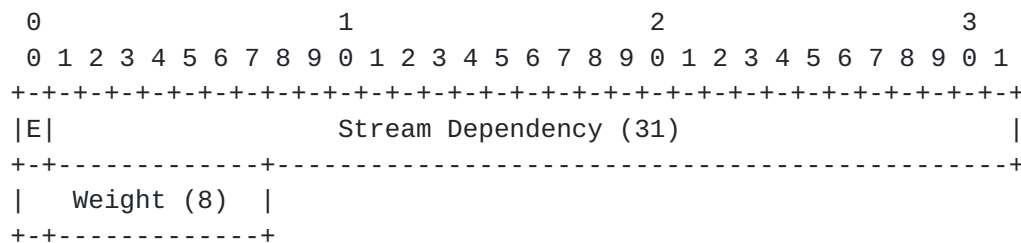
HEADERS frames MUST be associated with a stream. If a HEADERS frame is received whose stream identifier field is 0x0, the recipient MUST respond with a connection error ([Section 5.4.1](#)) of type `PROTOCOL_ERROR`.

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, including closed streams. 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)", "half closed (remote)", or "closed" 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 on the current stream. For a stream that is in the "half closed (remote)" or "closed" - state, this frame can only affect processing of the current stream and not frame transmission.

The `PRIORITY` frame is the only frame that can be sent for a stream in the "closed" state. This allows for the reprioritization of a group of dependent streams by altering the priority of a parent stream, which might be closed. However, a `PRIORITY` frame sent on a closed stream risks being ignored due to the peer having discarded priority state information for that stream.

#### [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.

```

      0               1               2               3
      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                               Error Code (32)                               |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+

```

`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 that affect how endpoints communicate, such as preferences and constraints on peer behavior. The SETTINGS frame is also used to acknowledge the receipt of those 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



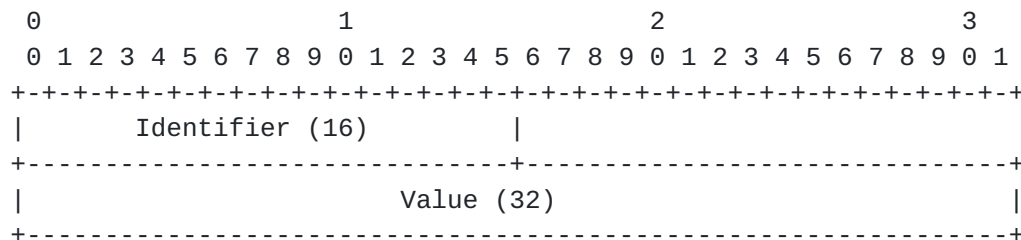
([Section 5.4.1](#)) of type FRAME\_SIZE\_ERROR. For more info, see Settings Synchronization ([Section 6.5.3](#)).

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

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

#### 6.5.1. SETTINGS Format

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



Setting Format

#### 6.5.2. Defined SETTINGS Parameters

The following parameters are defined:

SETTINGS\_HEADER\_TABLE\_SIZE (0x1): Allows the sender to inform the remote endpoint of the maximum size of the header compression table used to decode header blocks. The encoder can select any size equal to or less than this value by using signaling specific to the header compression format inside a header block. The initial value is 4,096 bytes.

SETTINGS\_ENABLE\_PUSH (0x2): 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 server push is permitted. Any value other than 0 or 1 MUST be treated as a connection error ([Section 5.4.1](#)) of type `PROTOCOL_ERROR`.

`SETTINGS_MAX_CONCURRENT_STREAMS (0x3)`: Indicates the maximum number of concurrent streams that the sender will allow. This limit is directional: it applies to the number of streams that the sender permits the receiver to create. Initially there is no limit to this value. It is recommended that this value be no smaller than 100, so as to not unnecessarily limit parallelism.

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

`SETTINGS_INITIAL_WINDOW_SIZE (0x4)`: 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`.

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

### [6.5.3](#). Settings Synchronization

Most values in `SETTINGS` benefit from or require an understanding of when the peer has received and applied the changed 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.

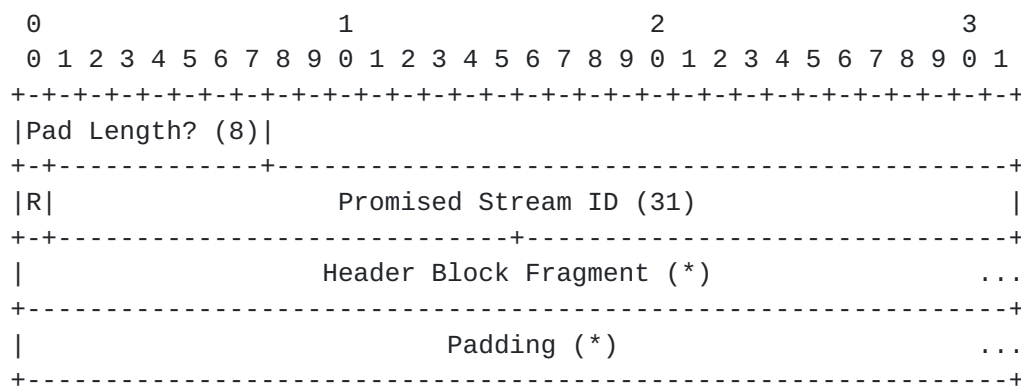




If the sender of a SETTINGS frame does not receive an acknowledgement within a reasonable amount of time, it MAY issue a connection error ([Section 5.4.1](#)) of type SETTINGS\_TIMEOUT.

## 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 Payload Format

The PUSH\_PROMISE frame payload has the following fields:

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

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

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 the order they are promised. The PUSH\_PROMISE only reserves stream identifiers for later use.

PUSH\_PROMISE MUST NOT be sent if the SETTINGS\_ENABLE\_PUSH setting of the peer endpoint is set to 0. An endpoint that has set this setting and has received acknowledgement MUST treat the receipt of a PUSH\_PROMISE frame as a connection error ([Section 5.4.1](#)) of type PROTOCOL\_ERROR.

Recipients of PUSH\_PROMISE frames can choose to reject promised streams by returning a RST\_STREAM referencing the promised stream identifier back to the sender of the PUSH\_PROMISE.

A PUSH\_PROMISE frame modifies the connection state in two ways. 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







## 6.8. GOAWAY

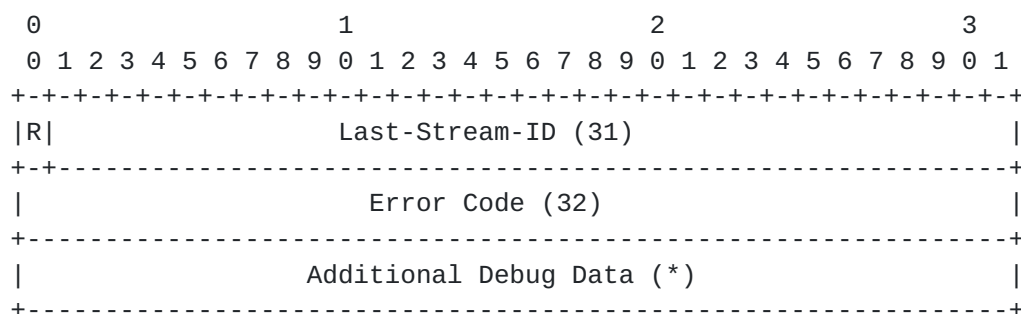
The GOAWAY frame (type=0x7) informs the remote peer to stop creating streams on this connection. GOAWAY can be sent by either the client or the server. Once sent, the sender will ignore frames sent on any new streams with identifiers higher than the included last stream identifier. Receivers of a GOAWAY frame MUST NOT open additional streams on the connection, although a new connection can be established for new streams.

The purpose of this frame is to allow an endpoint to gracefully stop accepting new streams, while still finishing processing of previously established streams. This enables administrative actions, like server maintenance.

There is an inherent race condition between an endpoint starting new streams and the remote sending a GOAWAY frame. To deal with this case, the GOAWAY contains the stream identifier of the last stream which was or might be processed on the sending endpoint in this connection. If the receiver of the GOAWAY has sent data on streams with a higher stream identifier than what is indicated in the GOAWAY frame, those streams are not or will not be processed. The receiver of the GOAWAY frame can treat the streams as though they had never been created at all, thereby allowing those streams to be retried later on a new connection.

Endpoints SHOULD always send a GOAWAY frame before closing a connection so that the remote can know whether a stream has been partially processed or not. For example, if an HTTP client sends a POST at the same time that a server closes a connection, the client cannot know if the server started to process that POST request if the server does not send a GOAWAY frame to indicate what streams it might have acted on.

An endpoint might choose to close a connection without sending 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 might have taken some action on, or might yet take action on. All streams up to and including the identified stream might have been processed in some way. The last stream identifier can be set to 0 if no streams were processed.

Note: In this context, "processed" means that some data from the stream was passed to some higher layer of software that might have taken some action as a result.

If a connection terminates without a GOAWAY frame, the last stream identifier is effectively the highest possible stream identifier.

On streams with lower or equal numbered identifiers that were not closed completely prior to the connection being closed, 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 indicates which streams could have been acted upon. Endpoints **MUST NOT** increase the value they send in the last stream identifier, since the peers might already have retried unprocessed requests on another connection.

A client that is unable to retry requests loses all requests that are in flight when the server closes the connection. This is especially true for intermediaries that might not be serving clients using HTTP/2. A server that is attempting to gracefully shut down a connection **SHOULD** send an initial GOAWAY frame with the last stream



identifier set to 2<sup>31</sup>-1 and a NO\_ERROR code. This signals to the client that a shutdown is imminent and that no further requests can be initiated. After waiting at least one round trip time, the server can send another GOAWAY frame with an updated last stream identifier. This ensures that a connection can be cleanly shut down without losing requests.

After sending a GOAWAY frame, the sender can discard frames for streams 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 frames. Frames that are exempt from flow control MUST be accepted and processed, unless the receiver is unable to assign resources to handling the frame. A receiver MAY respond with a stream error ([Section 5.4.2](#)) or connection error



([Section 5.4.1](#)) of type FLOW\_CONTROL\_ERROR if it is unable to accept a frame.

```

      0             1             2             3
      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|R|                               Window Size Increment (31)              |
+-----+

```

#### 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 capacity of the receiver.

Two flow control windows are applicable: the stream flow control window and the connection flow control window. The sender MUST NOT send a flow controlled frame with a length that exceeds the space



available in either of the flow control windows advertised by the receiver. Frames with zero length with the `END_STREAM` flag set (that is, an empty DATA frame) MAY be sent if there is no available space in either flow control window.

For flow control calculations, the 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.

#### **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 can only be changed using `WINDOW_UPDATE` frames.

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





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 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.

A SETTINGS frame cannot alter the connection flow control window.

An endpoint MUST treat a change to SETTINGS\_INITIAL\_WINDOW\_SIZE that causes any flow control window to exceed the maximum size as a connection error ([Section 5.4.1](#)) of type FLOW\_CONTROL\_ERROR.

### **6.9.3. Reducing the Stream Window Size**

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

After sending a SETTINGS frame that reduces the initial flow control window size, a receiver 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.

```

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

```

CONTINUATION Frame Payload

The CONTINUATION frame payload contains a header block fragment ([Section 4.3](#)).

The CONTINUATION frame defines the following flag:

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`.

The CONTINUATION frame changes the connection state as defined in [Section 4.3](#).

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

A CONTINUATION frame MUST be preceded by a HEADERS, PUSH\_PROMISE or CONTINUATION frame without the END\_HEADERS flag set. A recipient that observes violation of this rule MUST respond with a connection error ([Section 5.4.1](#)) of type `PROTOCOL_ERROR`.

## 7. Error Codes

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



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

The following error codes are defined:

**NO\_ERROR (0x0):** The associated condition is not as a result of an error. For example, a GOAWAY might include this code to indicate graceful shutdown of a connection.

**PROTOCOL\_ERROR (0x1):** The endpoint detected an unspecified protocol error. This error is for use when a more specific error code is not available.

**INTERNAL\_ERROR (0x2):** The endpoint encountered an unexpected internal error.

**FLOW\_CONTROL\_ERROR (0x3):** The endpoint detected that its peer violated the flow control protocol.

**SETTINGS\_TIMEOUT (0x4):** The endpoint sent a SETTINGS frame, but did not receive a response in a timely manner. See Settings Synchronization ([Section 6.5.3](#)).

**STREAM\_CLOSED (0x5):** The endpoint received a frame after a stream was half closed.

**FRAME\_SIZE\_ERROR (0x6):** The endpoint received a frame that was larger than the maximum size that it supports.

**REFUSED\_STREAM (0x7):** The endpoint refuses the stream prior to performing any application processing, see [Section 8.1.4](#) for details.

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

**COMPRESSION\_ERROR (0x9):** The endpoint is unable to maintain the header compression context for the connection.

**CONNECT\_ERROR (0xa):** The connection established in response to a CONNECT request ([Section 8.3](#)) was reset or abnormally closed.

**ENHANCE\_YOUR\_CALM (0xb):** The endpoint detected that its peer is exhibiting a behavior that might be generating excessive load.

**INADEQUATE\_SECURITY (0xc):** The underlying transport has properties that do not meet minimum security requirements (see [Section 9.2](#)).



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

## 8. HTTP Message Exchanges

HTTP/2 is intended to be as compatible as possible with current uses of HTTP. This means that, from the application perspective, the features of the protocol are largely 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 [[RFC7231](#)], Conditional Requests [[RFC7232](#)], Range Requests [[RFC7233](#)], Caching [[RFC7234](#)] and Authentication [[RFC7235](#)] are applicable to HTTP/2. Selected portions of HTTP/1.1 Message Syntax and Routing [[RFC7230](#)], 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 [[RFC7230](#)], [Section 3.2](#)), and
2. zero or more DATA frames containing the message payload (see [[RFC7230](#)], [Section 3.3](#)), and
3. optionally, one HEADERS frame, followed by zero or more CONTINUATION frames containing the trailer-part, if present (see [[RFC7230](#)], [Section 4.1.2](#)).

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

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

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 ([\[RFC7231\]](#), [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, thereby 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) [\[RFC2518\]](#) is



no longer necessary to ensure connection liveness, because HTTP/2 has a separate means of keeping the connection alive. The use of the 102 (Processing) status code for progress reporting has since been deprecated and is not retained.

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

- o An intermediary that translates HTTP/1.1 requests to HTTP/2 MUST generate a 100 (Continue) response if a received request includes an Expect header field with a "100-continue" token ([\[RFC7231\]](#), [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.
- o An intermediary that translates 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 [\[RFC7231\]](#), [Section 5.1.1](#) for specific requirements.
- o An intermediary that gateways HTTP/2 to HTTP/1.1 MUST discard all other 1xx informational responses.

### **[8.1.2](#). 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 [4].

While HTTP/1.x used the message start-line (see [\[RFC7230\]](#), [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 ':' character (ASCII 0x3a) for this purpose.

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 8.1.2.5](#)).

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 8.1.2.5](#)).

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.

#### **8.1.2.1. Request Header Fields**

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

- o The ":method" header field includes the HTTP method ([\[RFC7231\], Section 4](#)).
- o The ":scheme" header field includes the scheme portion of the target URI ([\[RFC3986\], Section 3.1](#)).

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

- o 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" schemes.

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 [\[RFC7230\], 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.

- o 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 8.1.2.5](#)).

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, and they MUST ignore and discard 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.

#### **[8.1.2.2](#). Response Header Fields**

A single ":status" header field is defined that carries the HTTP status code field (see [\[RFC7231, Section 6\]](#)). This header field MUST be included in all responses, otherwise the response is malformed ([Section 8.1.2.5](#)).

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

#### **[8.1.2.3](#). 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 [\[RFC7230, Section 3.2.2\]](#)), 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 directives in the first Cache-Control header field, and the third directive 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 not significant.

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.

#### **8.1.2.4. 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 [\[RFC7230\]](#), [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 8.1.2.3](#). When decoding, zero octets MUST be replaced with the cookie delimiter ("; ").

Therefore, the following sets of Cookie header fields are semantically equivalent, though the final form might appear in a different order after compression and decompression.

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

```
cookie: a=b\0c=d; e=f
```

```
cookie: a=b
```

```
cookie: c=d
```

```
cookie: e=f
```

#### **[8.1.2.5](#). 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 DATA frame payload lengths that form the body.

Intermediaries that process HTTP requests or responses (i.e., any intermediary not acting as a tunnel) 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 vulnerabilities.

#### **[8.1.3](#). Examples**

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

An HTTP GET request includes 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: "xyzzzy"              ==>  + END_STREAM
Expires: Thu, 23 Jan ...    + END_HEADERS
                             :status = 304
                             etag = "xyzzzy"
                             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
                              content-type = image/jpeg
{binary data}

CONTINUATION
+ END_HEADERS
  host = 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
                              content-type = image/jpeg
                              content-length = 123
{binary data}

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

#### 8.1.4. 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:

- o 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.
- o 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 [\[RFC7234\]](#), [Section 3](#)); promised requests **MUST** be safe (see [\[RFC7231\]](#), [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 8.1.2.1](#)). 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 8.1.2.2](#)) 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 ([\[RFC7231\]](#), [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 8.1.2.1](#)), with a few differences. Specifically:

- o The ":method" header field is set to "CONNECT".
- o The ":scheme" and ":path" header fields MUST be omitted.
- o 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 [\[RFC7230\]](#), [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 [\[RFC7231\]](#), [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 host and port pair, where host is derived from a URI, a selected alternative service [\[ALT-SVC\]](#), or a configured proxy.



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

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

Servers are encouraged to maintain open connections for as long as possible, but are permitted to terminate idle connections if necessary. When either endpoint chooses to close the transport-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.

#### **[9.1.1](#). Connection Reuse**

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](#)). For "http" resources, this depends on the host having resolved to the same IP address.

For "https" resources, connection reuse additionally depends on having a certificate that is valid for the host in the URI. That is the use of server certificate with multiple "subjectAltName" attributes, or names with wildcards. For example, a certificate with a "subjectAltName" of "\*.example.com" might permit the use of the same connection for "a.example.com" and "b.example.com".

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

A server that does not wish clients to reuse connections can indicate that it is not authoritative for a request by sending a 421 (Not Authoritative) status code in response to request (see [Section 9.1.2](#)).



### **9.1.2. The 421 (Not Authoritative) Status Code**

The 421 (Not Authoritative) status code indicates that the current origin server is not authoritative for the requested resource, in the sense of [\[RFC7230\]](#), [Section 9.1](#) (see also [Section 10.1](#)).

Clients receiving a 421 (Not Authoritative) response from a server MAY retry the request - whether the request method is idempotent or not - over a different connection. This is possible if a connection is reused ([Section 9.1.1](#)) or if an alternative service is selected ([\[ALT-SVC\]](#)).

This status code MUST NOT be generated by proxies.

A 421 response is cacheable by default; i.e., unless otherwise indicated by the method definition or explicit cache controls (see [Section 4.2.2 of \[RFC7234\]](#)).

## **9.2. Use of TLS Features**

Implementations of HTTP/2 MUST support TLS 1.2 [\[TLS12\]](#) for HTTP/2 over TLS. The general TLS usage guidance in [\[TLSBCP\]](#) SHOULD be followed, with some additional restrictions that are specific to HTTP/2.

### **9.2.1. TLS Features**

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.

The TLS implementation MUST disable renegotiation. An endpoint MUST treat a TLS renegotiation as a connection error ([Section 5.4.1](#)) of type `PROTOCOL_ERROR`. Note that disabling renegotiation can result in long-lived connections becoming unusable due to limits on the number of messages the underlying cipher suite can encipher.

A client MAY use renegotiation to provide confidentiality protection for client credentials offered in the handshake, but any renegotiation MUST occur prior to sending the connection preface. A



server SHOULD request a client certificate if it sees a renegotiation request immediately after establishing a connection.

This effectively prevents the use of renegotiation in response to a request for a specific protected resource. A future specification might provide a way to support this use case.

### **9.2.2. TLS Cipher Suites**

The set of TLS cipher suites that are permitted in HTTP/2 is restricted. HTTP/2 MUST only be used with cipher suites that have ephemeral key exchange, such as the ephemeral Diffie-Hellman (DHE) [[TLS12](#)] or the elliptic curve variant (ECDHE) [[RFC4492](#)]. Ephemeral key exchange MUST have a minimum size of 2048 bits for DHE or security level of 128 bits for ECDHE. Clients MUST accept DHE sizes of up to 4096 bits. HTTP MUST NOT be used with cipher suites that use stream or block ciphers. Authenticated Encryption with Additional Data (AEAD) modes, such as the Galois Counter Model (GCM) mode for AES [[RFC5288](#)] are acceptable.

Clients MAY advertise support of other cipher suites in order to allow for connection to servers that do not support HTTP/2 to complete without the additional latency imposed by using a separate connection for fallback.

An implementation SHOULD NOT negotiate a TLS connection for HTTP/2 without also negotiating a cipher suite that meets these requirements. Due to implementation limitations, it might not be possible to fail TLS negotiation. An endpoint MUST immediately terminate an HTTP/2 connection that does not meet these minimum requirements with a connection error ([Section 5.4.1](#)) of type INADEQUATE\_SECURITY.

## **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 [[RFC7230](#)], [Section 9.1](#). This relies on local name resolution for the "http" URI scheme, and the authenticated server identity for the "https" scheme (see [[RFC2818](#)], [Section 3](#)).





A client **MUST** discard responses 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 protection against cross protocol attacks. ALPN provides a positive indication that a server is willing to proceed with HTTP/2, which prevents attacks on other TLS-based protocols.

The encryption in TLS makes it difficult for attackers to control the data 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 in addition to the client connection preface 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 (ASCII 0xd) or line feed (ASCII 0xa) **MUST NOT** be translated verbatim by an intermediary, as stipulated in [\[RFC7230\]](#), [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.

The number of PUSH\_PROMISE frames is not constrained in the same fashion. A client that accepts server push SHOULD limit the number of streams it allows to be in the "reserved (remote)" state. Excessive number of server push streams can be treated as a stream error ([Section 5.4.2](#)) of type ENHANCE\_YOUR\_CALM.

Processing capacity cannot be guarded as effectively as state capacity.

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 or PRIORITY frames can be abused to cause an unnecessary waste of resources.



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 Section 8 of [[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.

#### **[10.5.1](#). Limits on Header Block Size**

A large header block ([Section 4.3](#)) can cause an implementation to commit a large amount of state. In servers and intermediaries, header fields that are critical to routing, such as ":authority", ":path", and ":scheme" are not guaranteed to be present early in the header block. In particular, values that are in the reference set cannot be emitted until the header block ends.

This can prevent streaming of the header fields to their ultimate destination, and forces the endpoint to buffer the entire header block. Since there is no hard limit to the size of a header block, an endpoint could be forced to exhaust available memory.

A server that receives a larger header block than it is willing to handle can send an HTTP 431 (Request Header Fields Too Large) status code [[RFC6585](#)]. A client can discard responses that it cannot process. The header block MUST be processed to ensure a consistent connection state, unless the connection is closed.



### **10.6. Use of Compression**

HTTP/2 enables greater use of compression for both header fields ([Section 4.3](#)) and entity bodies. 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.

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 or limiting compression might be preferable to padding as a countermeasure.

Padding can be used to obscure the exact size of frame content, and is provided to mitigate specific attacks within HTTP. For example, attacks where compressed content includes both attacker-controlled plaintext and secret data (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; similarly, 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 retain padding for DATA frames, but MAY drop padding for HEADERS and PUSH\_PROMISE frames. A valid reason for an intermediary to change the amount of padding of frames is to improve the protections that padding provides.

### **10.8. Privacy Considerations**

Several characteristics of HTTP/2 provide an observer an opportunity to correlate actions of a single client or server over time. 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 Section 1.8 of [[HTML5](#)].

## **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 frame types, settings, and error codes. These new registries are entered into a new "Hypertext Transfer Protocol (HTTP) 2 Parameters" section.

This document registers the "HTTP2-Settings" header field for use in HTTP; and the 421 (Not Authoritative) status code.

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



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

Protocol: HTTP/2 over TCP

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

Specification: This document

### **11.2. Frame Type Registry**

This document establishes a registry for HTTP/2 frame types codes. The "HTTP/2 Frame Type" registry manages an 8-bit space. The "HTTP/2 Frame Type" registry operates under either of the "IETF Review" or "IESG Approval" policies [[RFC5226](#)] for values between 0x00 and 0xef, with values between 0xf0 and 0xff being reserved for experimental use.

New entries in this registry require the following information:

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

Code: The 8-bit code assigned to the frame type.

Specification: A reference to a specification that includes a description of the frame layout, it's semantics and flags that the frame type uses, including any parts of the frame that are conditionally present based on the value of flags.

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

Frame Type	Code	Section
DATA	0x0	<a href="#">Section 6.1</a>
HEADERS	0x1	<a href="#">Section 6.2</a>
PRIORITY	0x2	<a href="#">Section 6.3</a>
RST_STREAM	0x3	<a href="#">Section 6.4</a>
SETTINGS	0x4	<a href="#">Section 6.5</a>
PUSH_PROMISE	0x5	<a href="#">Section 6.6</a>
PING	0x6	<a href="#">Section 6.7</a>
GOAWAY	0x7	<a href="#">Section 6.8</a>
WINDOW_UPDATE	0x8	<a href="#">Section 6.9</a>
CONTINUATION	0x9	<a href="#">Section 6.10</a>



### 11.3. Settings Registry

This document establishes a registry for HTTP/2 settings. The "HTTP/2 Settings" registry manages a 16-bit space. The "HTTP/2 Settings" registry operates under the "Expert Review" policy [RFC5226] for values in the range from 0x0000 to 0xffff, with values between 0xf000 and 0xffff being reserved for experimental use.

New registrations are advised to provide the following information:

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

**Code:** The 16-bit code assigned to the setting.

**Initial Value:** An initial value for the setting.

**Specification:** A stable reference to a specification that describes the use of the setting.

An initial set of setting registrations can be found in [Section 6.5.2](#).

Name	Code	Initial Value	Specification
HEADER_TABLE_SIZE	0x1	4096	<a href="#">Section 6.5.2</a>
ENABLE_PUSH	0x2	1	<a href="#">Section 6.5.2</a>
MAX_CONCURRENT_STREAMS	0x3	(infinite)	<a href="#">Section 6.5.2</a>
INITIAL_WINDOW_SIZE	0x4	65535	<a href="#">Section 6.5.2</a>

### 11.4. Error Code Registry

This document establishes a registry for HTTP/2 error codes. The "HTTP/2 Error Code" registry manages a 32-bit space. The "HTTP/2 Error Code" registry operates under the "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:

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



Code: The 32-bit error code value.

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

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

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

Name	Code	Description	Specification
NO_ERROR	0x0	Graceful shutdown	<a href="#">Section 7</a>
PROTOCOL_ERROR	0x1	Protocol error detected	<a href="#">Section 7</a>
INTERNAL_ERROR	0x2	Implementation fault	<a href="#">Section 7</a>
FLOW_CONTROL_ERROR	0x3	Flow control limits exceeded	<a href="#">Section 7</a>
SETTINGS_TIMEOUT	0x4	Settings not acknowledged	<a href="#">Section 7</a>
STREAM_CLOSED	0x5	Frame received for closed stream	<a href="#">Section 7</a>
FRAME_SIZE_ERROR	0x6	Frame size incorrect	<a href="#">Section 7</a>
REFUSED_STREAM	0x7	Stream not processed	<a href="#">Section 7</a>
CANCEL	0x8	Stream cancelled	<a href="#">Section 7</a>
COMPRESSION_ERROR	0x9	Compression state not updated	<a href="#">Section 7</a>
CONNECT_ERROR	0xa	TCP connection error for CONNECT method	<a href="#">Section 7</a>
ENHANCE_YOUR_CALM	0xb	Processing capacity exceeded	<a href="#">Section 7</a>
INADEQUATE_SECURITY	0xc	Negotiated TLS parameters not acceptable	<a href="#">Section 7</a>

### **11.5. 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.

#### **[11.6.](#) PRI Method Registration**

This section registers the "PRI" method in the HTTP Method Registry ([\[RFC7231\]](#), [Section 8.1](#)).

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.

#### **[11.7.](#) The 421 Not Authoritative HTTP Status Code**

This document registers the 421 (Not Authoritative) HTTP Status code in the Hypertext Transfer Protocol (HTTP) Status Code Registry ([\[RFC7231\]](#), [Section 8.2](#)).

Status Code: 421

Short Description: Not Authoritative

Specification: [Section 9.1.2](#) of this document

### **[12.](#) Acknowledgements**

This document includes substantial input from the following individuals:

- o 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).
- o Gabriel Montenegro and Willy Tarreau (Upgrade mechanism).



- o William Chan, Salvatore Loreto, Osama Mazahir, Gabriel Montenegro, Jitu Padhye, Roberto Peon, Rob Trace (Flow control).
- o Mike Bishop (Extensibility).
- o Mark Nottingham, Julian Reschke, James Snell, Jeff Pinner, Mike Bishop, Herve Ruellan (Substantial editorial contributions).
- o Alexey Melnikov was an editor of this document during 2013.
- o A substantial proportion of Martin's contribution was supported by Microsoft during his employment there.

## **13. References**

### **13.1. Normative References**

#### [COMPRESSION]

Ruellan, H. and R. Peon, "HPACK - Header Compression for HTTP/2", [draft-ietf-httpbis-header-compression-08](#) (work in progress), June 2014.

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

[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.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", [RFC 7230](#), June 2014.



- [RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", [RFC 7231](#), June 2014.
- [RFC7232] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests", [RFC 7232](#), June 2014.
- [RFC7233] Fielding, R., Ed., Lafon, Y., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Range Requests", [RFC 7233](#), June 2014.
- [RFC7234] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Caching", [RFC 7234](#), June 2014.
- [RFC7235] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Authentication", [RFC 7235](#), June 2014.
- [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.

### **13.2. Informative References**

- [ALT-SVC] Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", [draft-ietf-httpbis-alt-svc-01](#) (work in progress), April 2014.
- [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>>.



- [HTML5] Berjon, R., Faulkner, S., Leithead, T., Doyle Navara, E., O'Connor, E., and S. Pfeiffer, "HTML5", W3C Candidate Recommendation CR-html5-20140204, February 2014, <<http://www.w3.org/TR/2014/CR-html5-20140204/>>.
- Latest version available at [5].
- [RFC1323] Jacobson, V., Braden, B., and D. Borman, "TCP Extensions for High Performance", [RFC 1323](#), May 1992.
- [RFC2518] Goland, Y., Whitehead, E., Faizi, A., Carter, S., and D. Jensen, "HTTP Extensions for Distributed Authoring -- WEBDAV", [RFC 2518](#), February 1999.
- [RFC3749] Hollenbeck, S., "Transport Layer Security Protocol Compression Methods", [RFC 3749](#), May 2004.
- [RFC4492] Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., and B. Moeller, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)", [RFC 4492](#), May 2006.
- [RFC5288] Salowey, J., Choudhury, A., and D. McGrew, "AES Galois Counter Mode (GCM) Cipher Suites for TLS", [RFC 5288](#), August 2008.
- [RFC6585] Nottingham, N. and R. Fielding, "Additional HTTP Status Codes", [RFC 6585](#), April 2012.
- [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.

### **13.3. URIs**

- [1] <https://www.iana.org/assignments/message-headers>
- [2] <https://groups.google.com/forum/?fromgroups#!topic/spdy-dev/cfUef2gL3iU>
- [3] <https://tools.ietf.org/html/draft-montenegro-httpbis-http2-fc-principles-01>





**Appendix A. Change Log (to be removed by RFC Editor before publication)****A.1. Since [draft-ietf-httpbis-http2-12](#)**

Restored extensibility options.

Restricting TLS cipher suites to AEAD only.

Removing Content-Encoding requirements.

Permitting the use of PRIORITY after stream close.

Removed ALTSVC frame.

Removed BLOCKED frame.

Reducing the maximum padding size to 256 octets; removing padding from CONTINUATION frames.

Removed per-frame GZIP compression.

**A.2. Since [draft-ietf-httpbis-http2-11](#)**

Added BLOCKED frame (at risk).

Simplified priority scheme.

Added DATA per-frame GZIP compression.

**A.3. 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.4. 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.5. Since [draft-ietf-httpbis-http2-08](#)**

Added cookie crumbling for more efficient header compression.

Added header field ordering with the value-concatenation mechanism.

**A.6. Since [draft-ietf-httpbis-http2-07](#)**

Marked draft for implementation.

**A.7. 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.8. Since [draft-ietf-httpbis-http2-05](#)**

Marking the draft ready for implementation.

Renumbering END\_PUSH\_PROMISE flag.

Editorial clarifications and changes.

**A.9. 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.10. 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.11. 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.12. 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.13. 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 [6].

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 [7].

**A.14. Since [draft-mbelshe-httpbis-spy-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](mailto:mbelshe@chromium.org)

Roberto Peon

Google, Inc

EMail: [fenix@google.com](mailto:fenix@google.com)

Martin Thomson (editor)

Mozilla

331 E Evelyn Street

Mountain View, CA 94041

US

EMail: [martin.thomson@gmail.com](mailto:martin.thomson@gmail.com)

