HTTPbis Working Group
Internet-Draft
Intended status: Standards Track
Expires: November 30, 2013

M. Belshe
Twist
R. Peon
Google, Inc
M. Thomson, Ed.
Microsoft
A. Melnikov, Ed.
Isode Ltd

May 29, 2013

Hypertext Transfer Protocol version 2.0
draft-ietf-httpbis-http2-03

Abstract

This specification describes an optimized expression of the syntax of the Hypertext Transfer Protocol (HTTP). The HTTP/2.0 encapsulation enables more efficient use of network resources and reduced perception of latency by allowing header field compression and multiple concurrent messages on the same connection. It also introduces unsolicited push of representations from servers to clients.

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

Editorial Note (To be removed by RFC Editor)

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

Working Group information and related documents can be found at <http://tools.ietf.org/wg/httpbis/> (Wiki) and <https://github.com/http2/http2-spec> (source code and issues tracker).

The changes in this draft are summarized in Appendix A.1.

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 November 30, 2013.

Copyright Notice

Copyright (c) 2013 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

1. Introduction 5
  1.1. Document Organization 5
  1.2. Conventions and Terminology 6
2. Starting HTTP/2.0 6
  2.1. HTTP/2.0 Version Identification 7
  2.2. Starting HTTP/2.0 for "http:" URIs 8
  2.3. Starting HTTP/2.0 for "https:" URIs 8
  2.4. Starting HTTP/2.0 with Prior Knowledge 9
3. HTTP/2.0 Framing Layer 9
  3.1. Connection 9
  3.2. Connection Header 9
  3.3. Framing 10
    3.3.1. Frame Header 10
    3.3.2. Frame Size 12
  3.4. Streams 12
    3.4.1. Stream Creation 13

3.4.2. Stream priority 13
3.4.3. Stream half-close 14
3.4.4. Stream close 14
3.5. Error Handling 15
3.5.1. Connection Error Handling 15

3.5.2. Stream Error Handling 16
3.5.3. Error Codes 16
3.6. Stream Flow Control 17
3.6.1. Flow Control Principles 17
3.6.2. Appropriate Use of Flow Control 18
3.7. Header Blocks 19
3.8. Frame Types 19
3.8.1. DATA Frames 20
3.8.2. HEADERS+PRIORITY 20
3.8.3. RST_STREAM 21
3.8.4. SETTINGS 21
3.8.5. PUSH_PROMISE 25
3.8.6. PING 26
3.8.7. GOAWAY 26
3.8.8. HEADERS 28
3.8.9. WINDOW_UPDATE 29
4. HTTP Message Exchanges 32
4.1. Connection Management 32
4.2. HTTP Request/Response 33
4.2.1. HTTP Header Fields and HTTP/2.0 Headers 33
4.2.2. Request 33
4.2.3. Response 34
4.3. Server Push Transactions 35
4.3.1. Server implementation 36
4.3.2. Client implementation 37
5. Design Rationale and Notes 38
5.1. Separation of Framing Layer and Application Layer 38
5.2. Error handling - Framing Layer 39
5.3. One Connection per Domain 39
5.4. Fixed vs Variable Length Fields 39
5.5. Server Push 40
6. Security Considerations 40
6.1. Server Authority and Same-Origin 40
6.2. Cross-Protocol Attacks 40
6.3. Cacheability of Pushed Resources 41
7. Privacy Considerations 41

7.1. Long Lived Connections 41
7.2. SETTINGS frame 41
8. IANA Considerations 42
8.1. Frame Type Registry 42
8.2. Error Code Registry 43
8.3. Settings Registry 43
9. Acknowledgements 44
10. References 44
10.1. Normative References 44
10.2. Informative References 45
Appendix A. Change Log (to be removed by RFC Editor before
publication) 46

A.1. Since draft-ietf-httpbis-http2-02 46
A.2. Since draft-ietf-httpbis-http2-01 46
A.3. Since draft-ietf-httpbis-http2-00 47
A.4. Since draft-mbelshe-httpbis-spdy-00 47

1. Introduction

The Hypertext Transfer Protocol (HTTP) is a wildly successful protocol. However, the HTTP/1.1 message encapsulation ([HTTP-p1], Section 3) is optimized for implementation simplicity and accessibility, not application performance. As such it has several characteristics that have a negative overall effect on application performance.

In particular, HTTP/1.0 only allows one request to be delivered at a time on a given connection. HTTP/1.1 pipelining only partially addressed request concurrency, and is not widely deployed. Therefore, clients that need to make many requests (as is common on the Web) typically use multiple connections to a server in order to reduce perceived latency.

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

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

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

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

1.1. Document Organization

The HTTP/2.0 Specification is split into three parts: starting HTTP/2.0 (Section 2), which covers how a HTTP/2.0 connection is initiated; a framing layer (Section 3), which multiplexes a single TCP connection into independent frames of various types; and an HTTP layer (Section 4), which specifies the mechanism for expressing HTTP interactions using the framing layer. While some of the framing layer concepts are isolated from HTTP, building a generic framing

layer has not been a goal. The framing layer is tailored to the needs of the HTTP protocol and server push.

1.2. 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 connection.

connection: A transport-level connection between two endpoints.

endpoint: Either the client or server of the connection.

frame: The smallest unit of communication within an HTTP/2.0 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 connection.

connection error: An error on the HTTP/2.0 connection.

stream: A bi-directional flow of frames across a virtual channel within the HTTP/2.0 connection.

stream error: An error on the individual HTTP/2.0 stream.

2. Starting HTTP/2.0

HTTP/2.0 uses the same "http:" and "https:" URI schemes used by HTTP/1.1. 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/2.0.

The means by which support for HTTP/2.0 is determined is different for "http" and "https" URIs. Discovery for "https:" URIs is described in Section 2.3. Discovery for "http" URIs is described here.

2.1. HTTP/2.0 Version Identification

The protocol defined in this document is identified using the string "HTTP/2.0". This identification is used in the HTTP/1.1 Upgrade header field, in the TLS application layer protocol negotiation extension [TLSALPN] field and other places where protocol identification is required.

Negotiating "HTTP/2.0" implies the use of the transport, security, framing and message semantics described in this document.

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

Only implementations of the final, published RFC can identify themselves as "HTTP/2.0". Until such an RFC exists, implementations MUST NOT identify themselves using "HTTP/2.0".

Examples and text throughout the rest of this document use "HTTP/2.0" 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 "-draft-" and the corresponding draft number to the identifier before the separator ('/'). For example, draft-ietf-httpbis-http2-03 is identified using the string "HTTP-draft-03/2.0".

Non-compatible experiments that are based on these draft versions MUST instead replace the string "draft" with a different identifier. For example, an experimental implementation of packet mood-based encoding based on draft-ietf-httpbis-http2-07 might identify itself as "HTTP-emo-07/2.0". Note that any label MUST conform to the "token" syntax defined in Section 3.2.6 of [HTTP-p1]. Experimenters are encouraged to coordinate their experiments on the ietf-http-wg@w3.org mailing list.

2.2. Starting HTTP/2.0 for "http:" URIs

A client that makes a request to an "http:" URI without prior knowledge about support for HTTP/2.0 uses the HTTP Upgrade mechanism (Section 6.7 of [HTTP-p1]). The client makes an HTTP/1.1 request that includes an Upgrade header field identifying HTTP/2.0.

For example:

```
GET /default.htm HTTP/1.1
Host: server.example.com
Connection: Upgrade
Upgrade: HTTP/2.0
```

A server that does not support HTTP/2.0 can respond to the request as though the Upgrade header field were absent:

```
HTTP/1.1 200 OK
Content-length: 243
Content-type: text/html
      ...
```

A server that supports HTTP/2.0 can accept the upgrade with a 101 (Switching Protocols) status code. After the empty line that terminates the 101 response, the server can begin sending HTTP/2.0 frames. These frames MUST include a response to the request that initiated the Upgrade.

```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: HTTP/2.0

[ HTTP/2.0 connection ...
```

The first HTTP/2.0 frame sent by the server is a SETTINGS frame (Section 3.8.4). Upon receiving the 101 response, the client sends a connection header (Section 3.2), which includes a SETTINGS frame.

2.3. Starting HTTP/2.0 for "https:" URIs

A client that makes a request to an "https:" URI without prior knowledge about support for HTTP/2.0 uses TLS [RFC5246] with the application layer protocol negotiation extension [TLSALPN].

Once TLS negotiation is complete, both the client and the server send a connection header (Section 3.2).

2.4. Starting HTTP/2.0 with Prior Knowledge

A client can learn that a particular server supports HTTP/2.0 by other means. A client MAY immediately send HTTP/2.0 frames to a server that is known to support HTTP/2.0. This only affects the resolution of "http:" URIs, servers supporting HTTP/2.0 are required to support protocol negotiation in TLS [TLSALPN] for "https:" URIs.

Prior support for HTTP/2.0 is not a strong signal that a given server will support HTTP/2.0 for future connections. It is possible for server configurations to change or for configurations to differ between instances in clustered server. Interception proxies (a.k.a. "transparent" proxies) are another source of variability.

3. HTTP/2.0 Framing Layer

3.1. Connection

The HTTP/2.0 connection is an Application Level protocol running on top of a TCP connection ([RFC0793]). The client is the TCP connection initiator.

HTTP/2.0 connections are persistent. That is, for best performance, it is expected a 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.

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 MUST first send a GOAWAY (Section 3.8.7) frame so that both endpoints can reliably determine whether previously sent frames have been processed and gracefully complete or terminate any necessary remaining tasks.

3.2. Connection Header

Upon establishment of a TCP connection and determination that HTTP/2.0 will be used by both peers to communicate, each endpoint MUST send a connection header as a final confirmation and to establish the default parameters for the HTTP/2.0 connection.

The client connection header is a sequence of 24 octets (in hex notation)

464f4f202a20485454502f322e300d0a0d0a42410d0a0d0a

(the string "FOO * HTTP/2.0\r\n\r\nBA\r\n\r\n") followed by a SETTINGS frame (Section 3.8.4). The client sends the client connection header immediately upon receipt of a 101 Switching Protocols response (indicating a successful upgrade), or after receiving a TLS Finished message from the server. If starting an HTTP/2.0 connection with prior knowledge of server support for the protocol, the client connection header is sent upon connection establishment.

> The client connection header 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 header consists of just a SETTINGS frame (Section 3.8.4) that MUST be the first frame the server sends in the HTTP/2.0 connection.

To avoid unnecessary latency, clients are permitted to send additional frames to the server immediately after sending the client connection header, without waiting to receive the server connection header. It is important to note, however, that the server connection header 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.

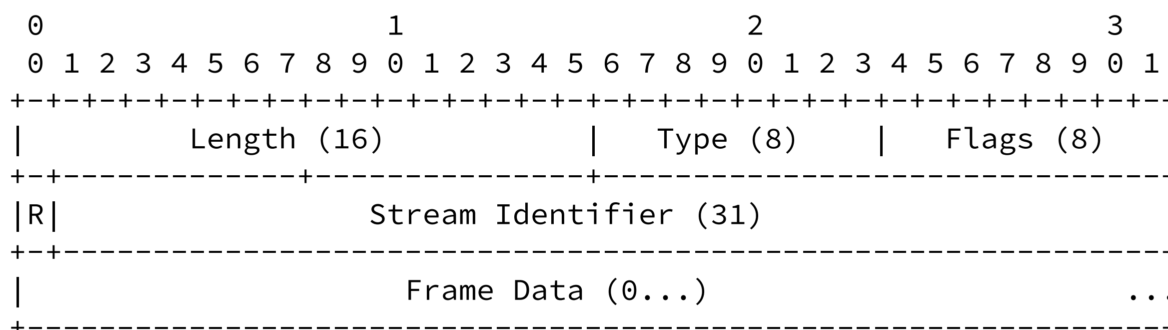
Clients and servers MUST terminate the TCP connection if either peer does not begin with a valid connection header. A GOAWAY frame (Section 3.8.7) MAY be omitted if it is clear that the peer is not using HTTP/2.0.

3.3. Framing

Once the HTTP/2.0 connection is established, clients and servers can begin exchanging frames.

3.3.1. Frame Header

HTTP/2.0 frames share a common base format consisting of an 8-byte header followed by 0 to 65535 bytes of data.


```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Length (16)           |   Type (8)    |   Flags (8)   |
+-+-------------+---------------+-------------------------------+
|R|                 Stream Identifier (31)                      |
+-+-------------------------------------------------------------+
|                   Frame Data (0...)                         ...
+---------------------------------------------------------------+
```

Frame Header

The fields of the frame header are defined as:

Length: The length of the frame data expressed as an unsigned 16-bit integer. The 8 bytes of the frame header are not included in this value.

Type: The 8-bit type of the frame. The frame type determines how the remainder of the frame header and data are interpreted. Implementations MUST ignore unsupported and unrecognized frame types.

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

The least significant bit (0x1) - the FINAL bit - is defined for all frame types as an indication that this frame is the last the endpoint will send for the identified stream. Setting this flag causes the stream to enter the half-closed state (Section 3.4.3). Implementations MUST process the FINAL bit for all frames whose stream identifier field is not 0x0. The FINAL bit MUST NOT be set

on frames that use a stream identifier of 0.

The remaining flags can be 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 3.4.1). A 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 remaining frame data is dependent

entirely on the frame type.

3.3.2. Frame Size

Implementations with limited resources might not be capable of processing large frame sizes. Such implementations MAY choose to place additional limits on the maximum frame size. However, all implementations MUST be capable of receiving and processing frames containing at least 8192 octets of data. [[anchor6: Ed. Question: Does this minimum include the 8-byte header or just the frame data?]]

An implementation MUST terminate a stream immediately if it is unable to process a frame due it's size. This is done by sending an RST_STREAM frame (Section 3.8.3) containing the FRAME_TOO_LARGE error code.

[[anchor7: <https://github.com/http2/http2-spec/issues/28>: Need a way to signal the maximum frame size; no way to RST_STREAM on non-stream-related frames.]]

3.4. Streams

A "stream" is an independent, bi-directional sequence of frames exchanged between the client and server within an HTTP/2.0 connection. Streams have several important characteristics:

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

- Streams can be rejected or cancelled by either endpoint.

- Multiple types of frames can be sent by either endpoint within a single stream.

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

- Streams optionally carry a set of name-value header pairs that are expressed within the headers block of HEADERS+PRIORITY, HEADERS, or PUSH_PROMISE frames.

- A single HTTP/2.0 connection can contain multiple concurrently active streams, with either endpoint interleaving frames from multiple streams.

3.4.1. Stream Creation

There is no coordination or shared action between the client and server required to create a stream. Rather, new streams are established by sending a frame whose stream identifier field references a previously unused stream identifier.

All streams are identified by an unsigned 31-bit integer. Streams initiated by a client use odd numbered stream identifiers; those initiated by the server use even numbered stream identifiers. A stream identifier of zero MUST NOT be used to establish a new stream.

The identifier of a newly established stream MUST be numerically greater than all previously established streams from that endpoint within the HTTP/2.0 connection, unless the identifier has been reserved using a PUSH_PROMISE (Section 3.8.5) frame. An endpoint that receives an unexpected stream identifier MUST respond with a connection error (Section 3.5.1) of type PROTOCOL_ERROR.

A peer can limit the total number of concurrently active streams using the SETTINGS_MAX_CONCURRENT_STREAMS parameters within a SETTINGS frame. The maximum concurrent streams setting is specific to each endpoint and applies only to the peer. That 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. Peer endpoints MUST NOT exceed this limit. All concurrently active streams initiated by an endpoint, including streams that are half-open (Section 3.4.3) in any direction, count toward that endpoint's limit.

Stream identifiers cannot be reused within a connection. Long-lived connections can cause an endpoint to exhaust 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.

Either endpoint can request the early termination of an unwanted stream by sending an RST_STREAM frame (Section 3.5.2) with an error code of either REFUSED_STREAM (if no frames have been processed) or CANCEL (if at least one frame has been processed). Such termination might not take effect immediately as the peer might have sent additional frames on the stream prior to receiving the termination request.

3.4.2. Stream priority

The endpoint establishing a new stream can assign a priority for the stream. Priority is represented as an unsigned 31-bit integer. 0 represents the highest priority and 2^31-1 represents the lowest

priority.

The purpose of this value is to allow the initiating endpoint to request that frames for the stream be processed with higher priority relative to any other concurrently active streams. That is, if an endpoint receives interleaved frames for multiple streams, the endpoint ought to make a best-effort attempt at processing frames for higher priority streams before processing those for lower priority streams.

Explicitly setting the priority for a stream does not guarantee any

particular processing order for the stream relative to any other stream. Nor is there is any mechanism provided by which the initiator of a stream can force or require a receiving endpoint to process frames from one stream before processing frames from another.

3.4.3. Stream half-close

When an endpoint sends a frame for a stream with the FINAL flag set, the stream is considered to be half-closed for that endpoint. Subsequent frames MUST NOT be sent by that endpoint for the half closed stream for the remaining duration of the HTTP/2.0 connection. When both endpoints have sent frames with the FINAL flag set, the stream is considered to be fully closed.

If an endpoint receives additional frames for a stream that was previously half-closed by the sending peer, the recipient MUST respond with a stream error (Section 3.5.2) of type STREAM_CLOSED.

An endpoint that has not yet half-closed a stream by sending the FINAL flag can continue sending frames on the stream.

It is not necessary for an endpoint to half-close a stream for which it has not sent any frames. This allows endpoints to use fully unidirectional streams that do not require explicit action or acknowledgement from the receiver.

3.4.4. Stream close

Streams can be terminated in the following ways:

Normal termination: Normal stream termination occurs when both client and server have half-closed the stream by sending a frame containing a FINAL flag (Section 3.3.1).

Half-close on unidirectional stream: A stream that only has frames sent in one direction can be tentatively considered to be closed once a frame containing a FINAL flag is sent. The active sender on the stream MUST be prepared to receive frames after closing the

stream.

Abrupt termination: Either peer can send a RST_STREAM control frame at any time to terminate an active stream. RST_STREAM contains an error code to indicate the reason for termination. A RST_STREAM indicates that the sender will transmit no further data on the stream and that the receiver is advised to cease transmission on it.

The sender of a RST_STREAM frame MUST allow for frames that have already been sent by the peer prior to the RST_STREAM being processed. If in-transit frames alter connection state, these frames cannot be safely discarded. See Stream Error Handling (Section 3.5.2) for more details.

TCP connection teardown: If the TCP connection is torn down while un-closed streams exist, then the endpoint MUST assume that the stream was abnormally interrupted and may be incomplete.

3.5. Error Handling

HTTP/2.0 framing permits two classes of error:

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

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

3.5.1. 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 MUST first send a GOAWAY (Section 3.8.7) frame 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 if the error is recurrent. Endpoints SHOULD send a GOAWAY frame when ending a connection, as long as circumstances permit it.

3.5.2. Stream Error Handling

A stream error is an error related to a specific stream identifier that does not affect processing of other streams at the framing layer.

An endpoint that detects a stream error sends a RST_STREAM (Section 3.8.3) frame 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 3.7)).

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.

3.5.3. Error Codes

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

Error codes share a common code space. Some error codes only apply to specific conditions and have no defined semantics in certain frame types.

The following error codes are defined:

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

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

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

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

INVALID_STREAM (4): The endpoint received a frame for an inactive stream.

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

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

REFUSED_STREAM (7): The endpoint is refusing the stream before processing its payload.

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

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

3.6. Stream 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.

HTTP/2.0 provides for flow control through use of the WINDOW_UPDATE (Section 3.8.9) frame type.

3.6.1. Flow Control Principles

Experience with TCP congestion control has shown that algorithms can evolve over time to become more sophisticated without requiring protocol changes. TCP congestion control and its evolution is clearly different from HTTP/2.0 flow control, though the evolution of TCP congestion control algorithms shows that a similar approach could be feasible for HTTP/2.0 flow control.

HTTP/2.0 stream flow control aims to allow for future improvements to

flow control algorithms without requiring protocol changes. Flow control in HTTP/2.0 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 octets they are prepared to receive on a stream. 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 preferences 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 65536 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 can be disabled by a receiver. A receiver can choose to either disable flow control for a stream or connection by declaring an infinite flow control limit.

7. HTTP/2.0 standardizes only the format of the window update frame (Section 3.8.9). This does not stipulate how a receiver decides

when to send this frame or the value that it sends. Nor does it specify how a sender chooses to send packets. Implementations are able to select any algorithm that suits their needs.

Implementations are also responsible for managing how requests and responses are sent based on priority; choosing how to avoid head of line blocking for requests; and managing the creation of new streams. Algorithm choices for these could interact with any flow control algorithm.

3.6.2. Appropriate Use of Flow Control

Flow control is defined to protect endpoints (client, server or intermediary) 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 SHOULD disable flow control for data that is being received. Note that flow control cannot be disabled for sending. Sending data is always subject to the flow control window advertised by the receiver.

Deployments with constrained resources (for example, memory) MAY employ flow control to limit the amount of memory a peer can consume. Note, however, that this can lead to suboptimal use of available network resources if flow control is enabled without knowledge of the bandwidth-delay product (see [RFC1323]).

Even with full awareness of the current bandwidth-delay product, implementation of flow control is difficult. However, it can ensure that constrained resources are protected without any reduction in connection utilization.

3.7. Header Blocks

The header block is found in the HEADERS, HEADERS+PRIORITY and PUSH_PROMISE frames. The header block consists of a set of header fields, which are name-value pairs. Headers are compressed using

black magic.

Compression of header fields is a work in progress, as is the format of this block.

The contents of header blocks MUST be processed by the compression context, even if stream has been reset or the frame is discarded. If header blocks cannot be processed, the receiver MUST treat the connection with a connection error (Section 3.5.1) of type COMPRESSION_ERROR.

3.8. Frame Types

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. Accordingly, while it is expected that new frame types will be introduced by extensions to this protocol, only frames defined by this document are permitted to alter the connection state.

3.8.1. DATA Frames

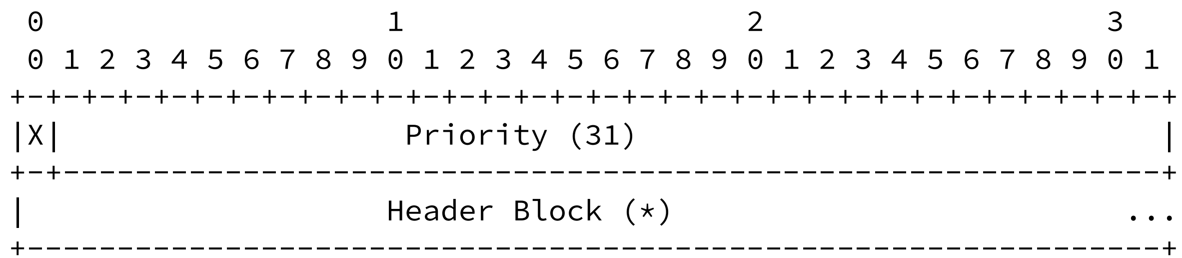
DATA frames (type=0x0) convey arbitrary, variable-length sequences of octets associated with a stream. One or more DATA frames are used, for instance, to carry HTTP request or response payloads.

The DATA frame does not define any type-specific flags.

DATA frames MUST be associated with a stream. If a DATA frame is received whose stream identifier field is 0x0, the recipient MUST respond with a connection error (Section 3.5.1) of type PROTOCOL_ERROR.

3.8.2. HEADERS+PRIORITY

The HEADERS+PRIORITY frame (type=0x1) allows the sender to set header fields and stream priority at the same time.



```
 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|X|                        Priority (31)                        |
+-+-------------------------------------------------------------+
|                        Header Block (*)                     ...
+---------------------------------------------------------------+
```

HEADERS+PRIORITY Frame Payload

The HEADERS+PRIORITY frame is identical to the HEADERS frame (Section 3.8.8), preceded by a single reserved bit and a 31-bit priority; see Section 3.4.2.

HEADERS+PRIORITY uses the same flags as the HEADERS frame, except that a HEADERS+PRIORITY frame with a CONTINUES bit MUST be followed by another HEADERS+PRIORITY frame. See HEADERS frame (Section 3.8.8) for any flags.

HEADERS+PRIORITY frames MUST be associated with a stream. If a HEADERS+PRIORITY frame is received whose stream identifier field is 0x0, the recipient MUST respond with a connection error (Section 3.5.1) of type PROTOCOL_ERROR.

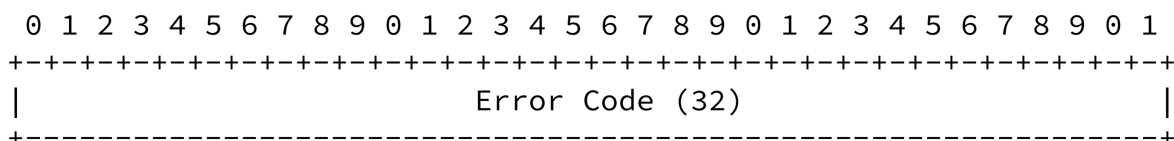
The HEADERS+PRIORITY frame modifies the connection state as defined

in Section 3.7.

3.8.3. 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. 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
```



RST_STREAM Frame Payload

The RST_STREAM frame contains a single unsigned, 32-bit integer identifying the error code (Section 3.5.3). The error code indicates why the stream is being terminated.

No type-flags are defined.

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 whose stream identifier field is 0x0 the recipient MUST respond with a connection error (Section 3.5.1) of type PROTOCOL_ERROR.

3.8.4. SETTINGS

The SETTINGS frame (type=0x4) conveys configuration parameters that affect how endpoints communicate. The parameters are either constraints on peer behavior or preferences.

SETTINGS frames MUST be sent at the start of a connection, and MAY be sent at any other time by either endpoint over the lifetime of the connection.

Implementations MUST support all of the settings defined by this specification and MAY support additional settings defined by extensions. Unsupported or unrecognized settings MUST be ignored. New settings MUST NOT be defined or implemented in a way that requires endpoints to understand then in order to communicate

successfully.

A SETTINGS frame is not required to include every defined setting; senders can include only those parameters for which it has accurate values and a need to convey. When multiple parameters are sent, they SHOULD be sent in order of numerically lowest ID to highest ID. A single SETTINGS frame MUST NOT contain multiple values for the same ID. If the receiver of a SETTINGS frame discovers multiple values for the same ID, it MUST ignore all values for that ID except the first one.

Over the lifetime of a connection, an endpoint MAY send multiple SETTINGS frames containing previously unspecified parameters or new values for parameters whose values have already been established. Only the most recent value provided setting value applies.

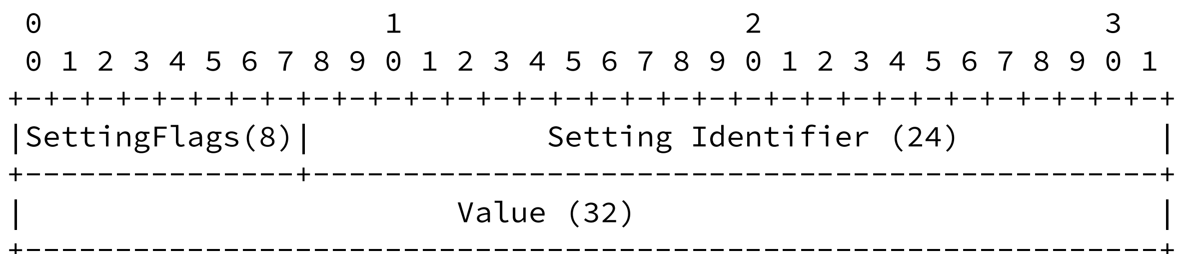
The SETTINGS frame defines the following flag:

CLEAR_PERSISTED (0x2): Bit 2 being set indicates a request to clear any previously persisted settings before processing the settings. Clients MUST NOT set this flag.

SETTINGS frames always apply to a connection, never a single stream. The stream identifier for a settings frame MUST be zero. If an endpoint receives a SETTINGS frame whose stream identifier field is anything other than 0x0, the endpoint MUST respond with a connection error (Section 3.5.1) of type PROTOCOL_ERROR.

3.8.4.1. Setting Format

The payload of a SETTINGS frame consists of zero or more settings. Each setting consists of an 8-bit flags field specifying per-item instructions, an unsigned 24-bit setting identifier, and an unsigned 32-bit value.



```
 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|SettingFlags(8)|             Setting Identifier (24)           |
+---------------+-----------------------------------------------+
|                         Value (32)                            |
+---------------------------------------------------------------+
```

Setting Format

Two flags are defined for the 8-bit flags field:

PERSIST_VALUE (0x1): Bit 1 (the least significant bit) being set indicates a request from the server to the client to persist this setting. A client MUST NOT set this flag.

PERSISTED (0x2): Bit 2 being set indicates that this setting is a persisted setting being returned by the client to the server. This also indicates that this setting is not a client setting, but a value previously set by the server. A server MUST NOT set this flag.

3.8.4.2. Setting Persistence

[[anchor12: Note that persistence of settings is under discussion in the WG and might be removed in a future version of this document.]]

A server endpoint can request that configuration parameters sent to a client in a SETTINGS frame are to be persisted by the client across HTTP/2.0 connections and returned to the server in any new SETTINGS frame the client sends to the server in the current connection or any future connections.

Persistence is requested on a per-setting basis by setting the PERSIST_VALUE flag (0x1).

Client endpoints are not permitted to make such requests. Servers MUST ignore any attempt by clients to request that a server persist configuration parameters.

Persistence of configuration parameters is done on a per-origin basis (see [RFC6454]). That is, when a client establishes a connection with a server, and the server requests that the client maintain persistent settings, the client SHOULD return the persisted settings on all future connections to the same origin, IP address and TCP port.

Whenever the client sends a SETTINGS frame in the current connection, or establishes a new connection with the same origin, persisted configuration parameters are sent with the PERSISTED flag (0x2) set for each persisted parameter.

Persisted settings accumulate until the server requests that all previously persisted settings are to be cleared by setting the CLEAR_PERSISTED (0x2) flag on the SETTINGS frame.

For example, if the server sends IDs 1, 2, and 3 with the FLAG_SETTINGS_PERSIST_VALUE in a first SETTINGS frame, and then sends IDs 4 and 5 with the FLAG_SETTINGS_PERSIST_VALUE in a subsequent SETTINGS frame, the client will return values for all 5 settings (1, 2, 3, 4, and 5 in this example) to the server.

3.8.4.3. Defined Settings

The following settings are defined:

SETTINGS_UPLOAD_BANDWIDTH (1): indicates the sender's estimated upload bandwidth for this connection. The value is an the integral number of kilobytes per second that the sender predicts as an expected maximum upload channel capacity.

SETTINGS_DOWNLOAD_BANDWIDTH (2): indicates the sender's estimated download bandwidth for this connection. The value is an integral number of kilobytes per second that the sender predicts as an expected maximum download channel capacity.

SETTINGS_ROUND_TRIP_TIME (3): indicates the sender's estimated round-trip-time for this connection. The round trip time is defined as the minimum amount of time to send a control frame from this client to the remote and receive a response. The value is represented in milliseconds.

SETTINGS_MAX_CONCURRENT_STREAMS (4): 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. By default there is no limit. It is recommended that this value be no smaller than 100, so as to not unnecessarily limit parallelism.

SETTINGS_CURRENT_CWND (5): indicates the sender's current TCP CWND value.

SETTINGS_DOWNLOAD_RETRANS_RATE (6): indicates the sender's retransmission rate (bytes retransmitted / total bytes transmitted).

SETTINGS_INITIAL_WINDOW_SIZE (7): indicates the sender's initial stream window size (in bytes) for new streams.

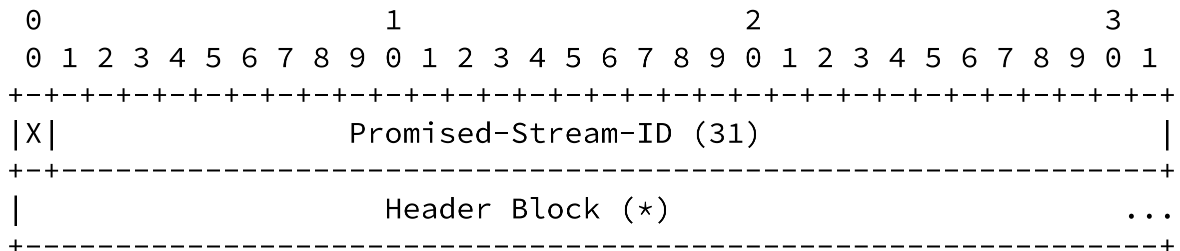
SETTINGS_FLOW_CONTROL_OPTIONS (10): indicates that streams directed to the sender will not be subject to flow control. The least significant bit (0x1) is set to indicate that new streams are not flow controlled. All other bits are reserved.

This setting applies to all streams, including existing streams.

These bits cannot be cleared once set, see Section 3.8.9.4.

3.8.5. 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 minimal set of headers that provide additional context for the stream. Section 4.3 contains a thorough description of the use of PUSH_PROMISE frames.



```
 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|X|                Promised-Stream-ID (31)                      |
+-+-------------------------------------------------------------+
|                 Header Block (*)                            ...
+---------------------------------------------------------------+
```

PUSH_PROMISE Payload Format

The payload of a PUSH_PROMISE includes a "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 3.4.1)).

PUSH_PROMISE frames MUST be associated with an existing stream. If the stream identifier field specifies the value 0x0, a recipient MUST respond with a connection error (Section 3.5.1) of type PROTOCOL_ERROR.

The state of promised streams is bound to the state of the original

associated stream on which the PUSH_PROMISE frame were sent. If the originating stream state changes to fully closed, all associated promised streams fully close as well. [[anchor13: Ed. Note: We need clarification on this point. How synchronized are the lifecycles of streams and associated promised streams?]]

PUSH_PROMISE uses the same flags as the HEADERS frame, except that a PUSH_PROMISE frame with a CONTINUES bit MUST be followed by another PUSH_PROMISE frame. See HEADERS frame (Section 3.8.8) for any flags.

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

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

The PUSH_PROMISE frame modifies the connection state as defined in Section 3.7.

3.8.6. PING

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

PING frames consist of an arbitrary, variable-length sequence of octets. Receivers of a PING send a response PING frame with the PONG flag set and precisely the same sequence of octets back to the sender as soon as possible.

Processing of PING frames SHOULD be performed with the highest priority if there are additional frames waiting to be processed.

The PING frame defines one type-specific flag:

PONG (0x2): Bit 2 being set indicates that this PING frame is a PING response. An endpoint MUST set this flag in PING responses. An endpoint MUST NOT respond to PING frames containing this flag.

PING frames are not associated with any individual stream. If a PING frame is received with a stream identifier field value other than 0x0, the recipient MUST respond with a connection error (Section 3.5.1) of type PROTOCOL_ERROR.

3.8.7. GOAWAY

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

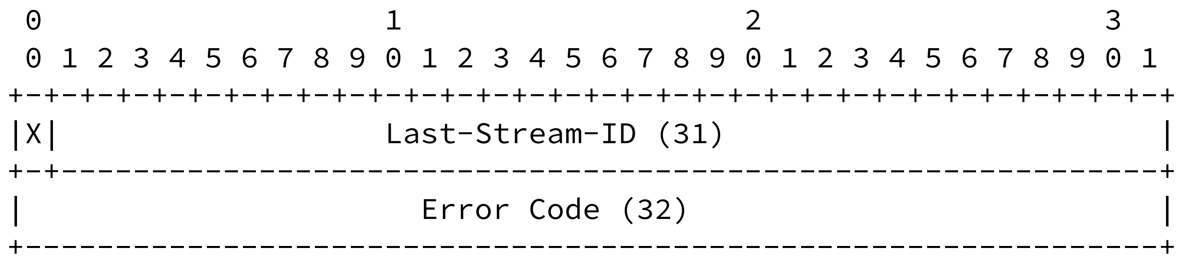
There is an inherent race condition between an endpoint starting new streams and the remote sending a GOAWAY frame. To deal with this case, the GOAWAY contains the stream identifier of the last stream

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

Endpoints should always send a GOAWAY frame before closing a connection so that the remote can know whether a stream has been partially processed or not. (For example, if an HTTP client sends a POST at the same time that a server closes a connection, the client cannot know if the server started to process that POST request if the server does not send a GOAWAY frame to indicate where it stopped working).

After sending a GOAWAY frame, the sender can ignore frames for new streams.

[[anchor14: Issue: connection state that is established by those "ignored" frames cannot be ignored without the state in the two peers becoming unsynchronized.]]



```
 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|X|                  Last-Stream-ID (31)                        |
+-+-------------------------------------------------------------+
|                      Error Code (32)                          |
+---------------------------------------------------------------+
```

GOAWAY Payload Format

The GOAWAY frame does not define any type-specific flags.

The GOAWAY frame applies to the connection, not a specific stream. The stream identifier MUST be zero.

The last stream identifier in the GOAWAY frame contains the highest numbered stream identifier for which the sender of the GOAWAY frame has received frames on and might have taken some action on. All streams up to and including the identified stream might have been processed in some way. The last stream identifier is set to 0 if no streams were processed.

> Note: In this case, "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.

On streams with lower or equal numbered identifiers that do not close 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 gracefully shut down a connection by sending a GOAWAY frame, maintaining the connection in an open state until all in-progress streams complete.

The last stream ID MUST be 0 if no streams were acted upon.

The GOAWAY frame also contains a 32-bit error code (Section 3.5.3) that contains the reason for closing the connection.

3.8.8. HEADERS

The HEADERS frame (type=0x8) provides header fields for a stream. Any number of HEADERS frames can may be sent on an existing stream at any time.

Additional type-specific flags for the HEADERS frame are:

CONTINUES (0x2): The CONTINUES bit indicates that this frame does not contain the entire payload necessary to provide a complete set of headers.

The payload for a complete set of headers is provided by a sequence of HEADERS frames, terminated by a HEADERS frame without the CONTINUES bit. Once the sequence terminates, the payload of all HEADERS frames are concatenated and interpreted as a single block.

A HEADERS frame that includes a CONTINUES bit MUST be followed by a HEADERS 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 3.5.1) of type PROTOCOL_ERROR.

The payload of a HEADERS frame contains a Headers Block (Section 3.7).

The HEADERS frame is associated with an existing stream. If a HEADERS frame is received with a stream identifier of 0x0, the recipient MUST respond with a stream error (Section 3.5.2) of type

PROTOCOL_ERROR.

The HEADERS frame changes the connection state as defined in Section 3.7.

3.8.9. WINDOW_UPDATE

The WINDOW_UPDATE frame (type=0x9) is used to implement flow control.

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

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

Flow control only applies to frames that are identified as being subject to flow control. Of the frame types defined in this document, this includes only DATA frame. Frames that are exempt from flow control MUST be accepted and processed, unless the receiver is unable to assign resources to handling the frame. A receiver MAY respond with a stream error (Section 3.5.2) or connection error (Section 3.5.1) of type FLOW_CONTROL_ERROR if it is unable accept a frame.

The following additional flags are defined for the WINDOW_UPDATE frame:

END_FLOW_CONTROL (0x2): Bit 2 being set indicates that flow control for the identified stream or connection has been ended; subsequent frames do not need to be flow controlled.

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.

The payload of a WINDOW_UPDATE frame is a 32-bit value indicating the additional number of bytes that the sender can transmit in addition to the existing flow control window. The legal range for this field is 1 to 2^31 - 1 (0x7fffffff) bytes; the most significant bit of this value is reserved.

3.8.9.1. The Flow Control Window

Flow control in HTTP/2.0 is implemented using a window kept by each sender on every stream. The flow control window is a simple integer value that indicates how many bytes of data the sender is permitted to transmit; as such, its size is a measure of the buffering capability of the receiver.

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

For flow control calculations, the 8 byte frame header is not counted.

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

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

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

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

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

3.8.9.2. Initial Flow Control Window Size

When a HTTP/2.0 connection is first established, new streams are created with an initial flow control window size of 65535 bytes. The connection flow control window is 65536 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 header.

Prior to receiving a SETTINGS frame that sets a value for SETTINGS_INITIAL_WINDOW_SIZE, a client 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 flow control windows that it maintains by the difference between the new value and the old value.

A change to SETTINGS_INITIAL_WINDOW_SIZE could 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 server sets the initial window size to be 16KB, and the client sends 64KB immediately on connection establishment, the client will recalculate the available flow control window to be -48KB 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.

3.8.9.3. 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.

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.

If a receiver decides to accept streams, both sides MUST recompute the available flow control window based on the initial window size

sent in the SETTINGS.

3.8.9.4. Ending Flow Control

After a receiver reads in a frame that marks the end of a stream (for example, a data stream with a FINAL flag set), it MUST cease transmission of WINDOW_UPDATE frames for that stream. A sender is not obligated to maintain the available flow control window for streams that it is no longer sending on.

Flow control can be disabled for all streams or the connection using the SETTINGS_FLOW_CONTROL_OPTIONS setting. An implementation that does not wish to perform flow control can use this in the initial SETTINGS exchange.

Flow control can be disabled for an individual stream or the overall connection by sending a WINDOW_UPDATE with the END_FLOW_CONTROL flag set. The payload of a WINDOW_UPDATE frame that has the END_FLOW_CONTROL flag set is ignored.

Flow control cannot be enabled again once disabled. Any attempt to re-enable flow control - by sending a WINDOW_UPDATE or by clearing the bits on the SETTINGS_FLOW_CONTROL_OPTIONS setting - MUST be rejected with a FLOW_CONTROL_ERROR error code.

4. HTTP Message Exchanges

HTTP/2.0 is intended to be as compatible as possible with current web-based applications. This means that, from the perspective of the server business logic or application API, the features of HTTP are unchanged. To achieve this, all of the application request and response header semantics are preserved, although the syntax of conveying those semantics has changed. Thus, the rules from HTTP/1.1 ([HTTP-p1], [HTTP-p2], [HTTP-p4], [HTTP-p5], [HTTP-p6], and [HTTP-p7]) apply with the changes in the sections below.

4.1. Connection Management

Clients SHOULD NOT open more than one HTTP/2.0 connection to a given origin ([RFC6454]) concurrently.

Note that it is possible for one HTTP/2.0 connection to be finishing (e.g. a GOAWAY frame has been sent, but not all streams have finished), while another HTTP/2.0 connection is starting.

4.2. HTTP Request/Response

4.2.1. HTTP Header Fields and HTTP/2.0 Headers

At the application level, HTTP uses name-value pairs in its header fields. Because HTTP/2.0 merges the existing HTTP header fields with HTTP/2.0 headers, there is a possibility that some HTTP applications already use a particular header field name. To avoid any conflicts, all header fields introduced for layering HTTP over HTTP/2.0 are prefixed with ":". ":" is not a valid sequence in HTTP/1.* header field naming, preventing any possible conflict.

4.2.2. Request

The client initiates a request by sending a HEADERS+PRIORITY frame. Requests that do not contain a body MUST set the FINAL flag, indicating that the client intends to send no further data on this stream, unless the server intends to push resources (see Section 4.3). HEADERS+PRIORITY frame does not contain the FINAL flag for requests that contain a body. The body of a request follows as a series of DATA frames. The last DATA frame sets the FINAL flag to indicate the end of the body.

The header fields included in the HEADERS+PRIORITY frame contain all of the HTTP header fields associated with an HTTP request. The definitions of these headers are largely unchanged relative to HTTP/1.1, with a few notable exceptions:

o The HTTP/1.1 request-line has been split into two separate header

fields named :method and :path, whose values specify the HTTP method for the request and the request-target, respectively. The HTTP-version component of the request-line is removed entirely from the headers.

- The host and optional port portions of the request URI (see [RFC3986], Section 3.2), is specified using the new :host header field. [[anchor21: Ed. Note: it needs to be clarified whether or not this replaces the existing HTTP/1.1 Host header.]]

- A new :scheme header field has been added to specify the scheme portion of the request-target (e.g. "https")

- All header field names MUST be lowercased, and the definitions of all header field names defined by HTTP/1.1 are updated to be all lowercase.

- The Connection, Host, Keep-Alive, Proxy-Connection, and Transfer-Encoding header fields are no longer valid and MUST not be sent.

All HTTP Requests MUST include the ":method", ":path", ":host", and ":scheme" header fields.

Header fields whose names begin with ":" (whether defined in this document or future extensions to this document) MUST appear before any other header fields.

If a client sends a HEADERS+PRIORITY frame that omits a mandatory header, the server MUST reply with a HTTP 400 Bad Request reply. [[anchor22: Ed: why PROTOCOL_ERROR on missing ":status" in the response, but HTTP 400 here?]]

If a server receives a request where the sum of the data frame payload lengths does not equal the size of the Content-Length header field, the server MUST return a 400 (Bad Request) error.

Although POSTs are inherently chunked, POST requests SHOULD also be accompanied by a Content-Length header field. First, it informs the server of how much data to expect, which the server can use to track overall progress and provide appropriate user feedback. More importantly, some HTTP server implementations fail to correctly process requests that omit the Content-Length header field. Many

existing clients send a Content-Length header field, and some server implementations have come to depend upon its presence.

A client provides priority in requests as a hint to the server. A server SHOULD attempt to provide responses to higher priority requests before lower priority requests. A server could send lower priority responses during periods that higher priority responses are unavailable to ensure better utilization of a connection.

If the server receives a data frame prior to a HEADERS+PRIORITY frame the server MUST treat this as a stream error (Section 3.5.2) of type PROTOCOL_ERROR.

4.2.3. Response

The server responds to a client request using the same stream identifier that was used by the request. An HTTP response begins with a HEADERS frame. An HTTP response body consists of a series of DATA frames. The last data frame contains a FINAL flag to indicate the end of the response. A response that contains no body (such as a 204 or 304 response) consists only of a HEADERS frame that contains the FINAL flag to indicate no further data will be sent on the stream.

The response status line is unfolded into name-value pairs like other HTTP header fields and must be present:

":status": The HTTP response status code (e.g. "200" or "200 OK")

All header field names starting with ":" (whether defined in this document or future extensions to this document) MUST appear before any other header fields.

All header field names MUST be all lowercase.

The Connection, Keep-Alive, Proxy-Connection, and Transfer-Encoding header fields are not valid and MUST not be sent.

Responses MAY be accompanied by a Content-Length header field for

advisory purposes. This allows clients to learn the full size of an entity prior to receiving all the data frames. This can help in, for example, reporting progress.

If a client receives a response where the sum of the data frame payload length does not equal the size of the Content-Length header field, the client MUST ignore the content length header field. [[anchor23: Ed: See <https://github.com/http2/http2-spec/issues/46>.]]

If a client receives a response with an absent or duplicated status header, the client MUST treat this as a stream error (Section 3.5.2) of type PROTOCOL_ERROR.

If the client receives a data frame prior to a HEADERS frame the client MUST treat this as a stream error (Section 3.5.2) of type PROTOCOL_ERROR.

Clients MUST support gzip compression. Regardless of the value of the Accept-Encoding header field, a server MAY send responses with gzip or deflate encoding. A compressed response MUST still bear an appropriate Content-Encoding header field.

4.3. Server Push Transactions

HTTP/2.0 enables a server to send multiple replies to a client for a single request. The rationale for this feature is that sometimes a server knows that it will need to send multiple resources in response to a single request. Without server push features, the client must first download the primary resource, then discover the secondary resource(s), and request them.

Server push is an optional feature. The

SETTINGS_MAX_CONCURRENT_STREAMS setting from the client limits the number of resources that can be concurrently pushed by a server. Server push can be disabled by clients that do not wish to receive pushed resources by advertising a SETTINGS_MAX_CONCURRENT_STREAMS SETTING (Section 3.8.4) of zero. This prevents servers from creating the streams necessary to push resources.

Clients receiving a pushed response MUST validate that the server is

authorized to push the resource using the same-origin policy ([RFC6454], Section 3). For example, a HTTP/2.0 connection to "example.com" is generally [[anchor24: Ed: weaselly use of "generally", needs better definition]] not permitted to push a response for "www.example.org".

A client that accepts pushed resources caches those resources as though they were responses to GET requests.

Pushing of resources avoids the round-trip delay, but also creates a potential race where a server can be pushing content which a client is in the process of requesting. The PUSH_PROMISE frame reduces the chances of this condition occurring, while retaining the performance benefit.

Pushed responses are associated with a request at the HTTP/2.0 framing layer. The PUSH_PROMISE is sent on the stream for the associated request, which allows a receiver to correlate the pushed resource with a request. The pushed stream inherits all of the request header fields from the associated stream with the exception of resource identification header fields (":host", ":scheme", and ":path"), which are provided as part of the PUSH_PROMISE frame.

Pushed resources always have an associated ":method" of "GET". A cache MUST store these inherited and implied request header fields with the cached resource.

4.3.1. Server implementation

A server pushes resources in association with a request from the client. Prior to closing the response stream, the server sends a PUSH_PROMISE for each resource that it intends to push. The PUSH_PROMISE includes header fields that allow the client to identify the resource (":scheme", ":host", and ":path").

A server can push multiple resources in response to a request, but all pushed resources MUST be promised on the response stream for the associated request. A server cannot send a PUSH_PROMISE on a new stream or a half-closed stream.

The server SHOULD include any header fields in a PUSH_PROMISE that

would allow a cache to determine if the resource is already cached (see [HTTP-p6], Section 4).

After sending a PUSH_PROMISE, the server commences transmission of a pushed resource. A pushed resource uses a server-initiated stream. The server sends frames on this stream in the same order as an HTTP response (Section 4.2.3): a HEADERS frame followed by DATA frames.

Many uses of server push are to send content that a client is likely to discover a need for based on the content of a response representation. To minimize the chances that a client will make a request for resources that are being pushed - causing duplicate copies of a resource to be sent by the server - a PUSH_PROMISE frame SHOULD be sent prior to any content in the response representation that might allow a client to discover the pushed resource and request it.

The server MUST only push resources that could have been returned from a GET request.

Note: A server does not need to have all response header fields available at the time it issues a PUSH_PROMISE frame. All remaining header fields are included in the HEADERS frame. The HEADERS frame MUST NOT duplicate header fields from the PUSH_PROMISE frames.

4.3.2. Client implementation

When fetching a resource the client has 3 possibilities:

1. the resource is not being pushed

2. the resource is being pushed, but the data has not yet arrived

3. the resource is being pushed, and the data has started to arrive

A client SHOULD NOT issue GET requests for a resource that has been promised. A client is instead advised to wait for the pushed resource to arrive.

When a client receives a PUSH_PROMISE frame from the server without a the ":host", ":scheme", and ":path" header fields, it MUST treat this as a stream error (Section 3.5.2) of type PROTOCOL_ERROR.

To cancel individual server push streams, the client can issue a stream error (Section 3.5.2) of type CANCEL. After receiving a PUSH_PROMISE frame, the client is able to cancel the pushed resource before receiving any frames on the promised stream. The server

ceases transmission of the pushed resource; if the server has not commenced transmission, it does not start.

To cancel all server push streams related to a request, the client may issue a stream error (Section 3.5.2) of type CANCEL on the associated-stream-id. By cancelling that stream, the server MUST immediately stop sending frames for any streams with in-association-to for the original stream. [[anchor27: Ed: Triggering side-effects on stream reset is going to be problematic for the framing layer. Purely from a design perspective, it's a layering violation. More practically speaking, the base request stream might already be removed. Special handling logic would be required.]]

A client can choose to time out pushed streams if the server does not provide the resource in a timely fashion. A stream error (Section 3.5.2) of type CANCEL can be used to stop a timed out push.

If the server sends a HEADERS frame containing header fields that duplicate values on a previous HEADERS or PUSH_PROMISE frames on the same stream, the client MUST treat this as a stream error (Section 3.5.2) of type PROTOCOL_ERROR.

If the server sends a HEADERS frame after sending a data frame for the same stream, the client MAY ignore the HEADERS frame. Ignoring the HEADERS frame after a data frame prevents handling of HTTP's trailing header fields (Section 4.1.1 of [HTTP-p1]).

5. Design Rationale and Notes

Authors' notes: The notes in this section have no bearing on the HTTP/2.0 protocol as specified within this document, and none of these notes should be considered authoritative about how the protocol works. However, these notes may prove useful in future debates about how to resolve protocol ambiguities or how to evolve the protocol going forward. They may be removed before the final draft.

5.1. Separation of Framing Layer and Application Layer

Readers may note that this specification sometimes blends the framing layer (Section 3) with requirements of a specific application - HTTP (Section 4). This is reflected in the request/response nature of the streams and the definition of the HEADERS which are very similar to HTTP, and other areas as well.

This blending is intentional - the primary goal of this protocol is to create a low-latency protocol for use with HTTP. Isolating the

two layers is convenient for description of the protocol and how it relates to existing HTTP implementations. However, the ability to

reuse the HTTP/2.0 framing layer is a non goal.

5.2. Error handling - Framing Layer

Error handling at the HTTP/2.0 layer splits errors into two groups: Those that affect an individual HTTP/2.0 stream, and those that do not.

When an error is confined to a single stream, but general framing is intact, HTTP/2.0 attempts to use the RST_STREAM as a mechanism to invalidate the stream but move forward without aborting the connection altogether.

For errors occurring outside of a single stream context, HTTP/2.0 assumes the entire connection is hosed. In this case, the endpoint detecting the error should initiate a connection close.

5.3. One Connection per Domain

HTTP/2.0 attempts to use fewer connections than other protocols have traditionally used. The rationale for this behavior is because it is very difficult to provide a consistent level of service (e.g. TCP slow-start), prioritization, or optimal compression when the client is connecting to the server through multiple channels.

Through lab measurements, we have seen consistent latency benefits by using fewer connections from the client. The overall number of packets sent by HTTP/2.0 can be as much as 40% less than HTTP. Handling large numbers of concurrent connections on the server also does become a scalability problem, and HTTP/2.0 reduces this load.

The use of multiple connections is not without benefit, however. Because HTTP/2.0 multiplexes multiple, independent streams onto a single stream, it creates a potential for head-of-line blocking problems at the transport level. In tests so far, the negative effects of head-of-line blocking (especially in the presence of packet loss) is outweighed by the benefits of compression and prioritization.

5.4. Fixed vs Variable Length Fields

HTTP/2.0 favors use of fixed length 32bit fields in cases where smaller, variable length encodings could have been used. To some, this seems like a tragic waste of bandwidth. HTTP/2.0 chooses the simple encoding for speed and simplicity.

The goal of HTTP/2.0 is to reduce latency on the network. The overhead of HTTP/2.0 frames is generally quite low. Each data frame

is only an 8 byte overhead for a 1452 byte payload (~0.6%). At the time of this writing, bandwidth is already plentiful, and there is a strong trend indicating that bandwidth will continue to increase. With an average worldwide bandwidth of 1Mbps, and assuming that a variable length encoding could reduce the overhead by 50%, the latency saved by using a variable length encoding would be less than 100 nanoseconds. More interesting are the effects when the larger encodings force a packet boundary, in which case a round-trip could be induced. However, by addressing other aspects of HTTP/2.0 and TCP interactions, we believe this is completely mitigated.

5.5. Server Push

A subtle but important point is that server push streams must be declared before the associated stream is closed. The reason for this is so that proxies have a lifetime for which they can discard information about previous streams. If a pushed stream could associate itself with an already-closed stream, then endpoints would not have a specific lifecycle for when they could disavow knowledge of the streams which went before.

6. Security Considerations

6.1. Server Authority and Same-Origin

This specification uses the same-origin policy ([RFC6454], Section 3) to determine whether an origin server is permitted to provide content.

A server that is contacted using TLS is authenticated based on the certificate that it offers in the TLS handshake (see [RFC2818], Section 3). A server is considered authoritative for an "https:"

resource if it has been successfully authenticated for the domain part of the origin of the resource that it is providing.

A server is considered authoritative for an "http:" resource if the connection is established to a resolved IP address for the domain in the origin of the resource.

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

6.2. Cross-Protocol Attacks

When using TLS, we believe that HTTP/2.0 introduces no new cross-protocol attacks. TLS encrypts the contents of all transmission (except the handshake itself), making it difficult for attackers to control the data which could be used in a cross-protocol attack.

[[anchor37: Issue: This is no longer true]]

6.3. Cacheability of Pushed Resources

Pushed resources are synthesized responses without an explicit request; the request for a pushed resource is synthesized from the request that triggered the push, plus resource identification information provided by the server. Request header fields are necessary for HTTP cache control validations (such as the Vary header field) to work. For this reason, caches MUST inherit request header fields from the associated stream for the push. This includes the Cookie header field.

Caching resources 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 resources for which an origin server is not authoritative are never cached or used.

7. Privacy Considerations

7.1. Long Lived Connections

HTTP/2.0 aims to keep connections open longer between clients and servers in order to reduce the latency when a user makes a request. The maintenance of these connections over time could be used to expose private information. For example, a user using a browser hours after the previous user stopped using that browser may be able to learn about what the previous user was doing. This is a problem with HTTP in its current form as well, however the short lived connections make it less of a risk.

7.2. SETTINGS frame

The HTTP/2.0 SETTINGS frame allows servers to store out-of-band transmitted information about the communication between client and server on the client. Although this is intended only to be used to reduce latency, renegade servers could use it as a mechanism to store

identifying information about the client in future requests.

Clients implementing privacy modes can disable client-persisted SETTINGS storage.

Clients MUST clear persisted SETTINGS information when clearing the cookies.

8. IANA Considerations

This document establishes registries for frame types, error codes and settings.

8.1. Frame Type Registry

This document establishes a registry for HTTP/2.0 frame types. The "HTTP/2.0 Frame Type" registry operates under the "IETF Review" policy [RFC5226].

Frame types are an 8-bit value. When reviewing new frame type registrations, special attention is advised for any frame type-specific flags that are defined. Frame flags can interact with existing flags and could prevent the creation of globally applicable flags.

Initial values for the "HTTP/2.0 Frame Type" registry are shown in Table 1.

Frame Type	Name	Flags
0	DATA	-
1	HEADERS+PRIORITY	-
3	RST_STREAM	-
4	SETTINGS	CLEAR_PERSISTED(2)
5	PUSH_PROMISE	-
6	PING	PONG(2)
7	GOAWAY	-
8	HEADERS	-
9	WINDOW_UPDATE	END_FLOW_CONTROL(2)

Table 1

8.2. Error Code Registry

This document establishes a registry for HTTP/2.0 error codes. The "HTTP/2.0 Error Code" registry manages a 32-bit space. The "HTTP/2.0 Error Code" registry operates under the "Expert Review" policy [RFC5226].

Registrations for error codes are required to include a description of the error code. An expert reviewer is advised to examine new registrations for possible duplication with existing error codes. Use of existing registrations is to be encouraged, but not mandated.

New registrations are advised to provide the following information:

Error Code: The 32-bit error code value.

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

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

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

An initial set of error code registrations can be found in Section 3.5.3.

8.3. Settings Registry

This document establishes a registry for HTTP/2.0 settings. The "HTTP/2.0 Settings" registry manages a 24-bit space. The "HTTP/2.0 Settings" registry operates under the "Expert Review" policy [RFC5226].

Registrations for settings are required to include a description of the setting. An expert reviewer is advised to examine new registrations for possible duplication with existing settings. Use of existing registrations is to be encouraged, but not mandated.

New registrations are advised to provide the following information:

Setting: The 24-bit setting value.

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

Flags: Any setting-specific flags that apply, including their value and semantics.

Description: A description of the setting. This might include the

range of values, any applicable units and how to act upon a value when it is provided.

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

An initial set of settings registrations can be found in Section 3.8.4.3.

9. 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 Mark Nottingham, Julian Reschke, James Snell (Editorial)

10. References

10.1. Normative References

[HTTP-p1] Fielding, R. and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", draft-ietf-httpbis-p1-messaging-22 (work in progress), February 2013.

[HTTP-p2] Fielding, R. and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", draft-ietf-httpbis-p2-semantics-22 (work in progress), February 2013.

[HTTP-p4] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests", draft-ietf-httpbis-p4-conditional-22 (work in progress), February 2013.

[HTTP-p5] Fielding, R., Ed., Lafon, Y., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Range Requests", draft-ietf-httpbis-p5-range-22 (work in progress), February 2013.

[HTTP-p6] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Caching", draft-ietf-httpbis-p6-cache-22 (work in progress), February 2013.

[HTTP-p7] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Authentication", draft-ietf-httpbis-p7-auth-22 (work in progress), February 2013.

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

[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.

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

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

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

[TLSALPN] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application Layer Protocol Negotiation Extension", draft-ietf-tls-applayerprotoneg-01 (work in progress), April 2013.

10.2. Informative References

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

[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>.

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

A.1. 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.2. 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.3. Since draft-ietf-httpbis-http2-00

Changed title throughout.

Removed section on Incompatibilities with SPDY draft#2.

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

Replaced abstract and introduction.

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

Removed unused references.

Added flow control principles (Section 3.6.1) based on <http://tools.ietf.org/html/draft-montenegro-httpbis-http2-fc-principles-01>.

A.4. Since draft-mbelshe-httpbis-spdy-00

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

Updated authors/editors list.

Added status note.

Authors' Addresses

Mike Belshe
Twist

EMail: mbelshe@chromium.org

Roberto Peon
Google, Inc

EMail: fenix@google.com

Martin Thomson (editor)
Microsoft
3210 Porter Drive
Palo Alto 94304
US

EMail: martin.thomson@skype.net

Alexey Melnikov (editor)
Isode Ltd
5 Castle Business Village
36 Station Road
Hampton, Middlesex TW12 2BX
UK

EMail: Alexey.Melnikov@isode.com