HTTPbis Working Group M. Belshe
Internet-Draft Twist
Intended status: Standards Track R. Peon
Expires: January 9, 2014 Google, Inc
M. Thomson, Ed.
Microsoft
A. Melnikov, Ed.
Isode Ltd
July 8, 2013

Hypertext Transfer Protocol version 2.0
draft-ietf-httpbis-http2-04

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.

This version of the draft has been marked for implementation. Interoperability testing will occur in the HTTP/2.0 interim in Hamburg, DE, starting 2013-08-05.

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.

This Internet-Draft will expire on January 9, 2014.

Copyright Notice

Table of Contents

1. Introduction

The Hypertext Transfer Protocol (HTTP) is a wildly successful protocol. However, the HTTP/1.1 message format ([HTTP-p1], Section 3) 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 3), which covers how a HTTP/2.0 connection is initiated; a framing layer (Section 4), which multiplexes a single TCP connection into independent frames of various types; and an HTTP layer (Section 8), 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. HTTP/2.0 Protocol Overview

HTTP/2.0 provides an optimized transport for HTTP semantics.

An HTTP/2.0 connection is an application level protocol running on top of a TCP connection ([RFC0793]). The client is the TCP

connection initiator.

This document describes the HTTP/2.0 protocol using a logical structure that is formed of three parts: framing, streams, and application mapping. This structure is provided primarily as an aid to specification, implementations are free to diverge from this structure as necessary.

2.1. HTTP Frames

HTTP/2.0 provides an efficient serialization of HTTP semantics. HTTP requests and responses are encoded into length-prefixed frames (see Section 4.1).

HTTP headers are compressed into a series of frames that contain header block fragments (see Section 4.3).

2.2. HTTP Multiplexing

HTTP/2.0 provides the ability to multiplex multiple HTTP requests and responses onto a single connection. Multiple requests or responses can be sent concurrently on a connection using streams (Section 5). In order to maintain independent streams, flow control and prioritization are necessary.

2.3. HTTP Semantics

HTTP/2.0 defines how HTTP requests and responses are mapped to streams (see Section 8) and introduces a new interaction model, server push (Section 8.2).

3. Starting HTTP/2.0

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

The means by which support for HTTP/2.0 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/2.0 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.

[[anchor6: 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.

3.2. 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. The HTTP/1.1 request MUST include an HTTP2-Settings (Section 3.2.1) header field.

For example:

```
GET /default.htm HTTP/1.1
Host: server.example.com
Connection: Upgrade, HTTP2-Settings
Upgrade: HTTP/2.0
HTTP2-Settings: <base64url encoding of HTTP/2.0 SETTINGS payload>
```

Requests that contain a request entity body MUST be sent in their entirety before the client can send HTTP/2.0 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/2.0, at the cost of an additional round trip.

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 accepts 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 6.5). Upon receiving the 101 response, the client sends a connection header (Section 3.5), which includes a SETTINGS frame.

The HTTP/1.1 request that is sent prior to upgrade is associated with stream 1 and is assigned the highest possible priority. 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/2.0 connection, stream 1 is used for the response.

3.2.1. HTTP2-Settings Header Field

A client that upgrades from HTTP/1.1 to HTTP/2.0 MUST include an "HTTP2-Settings" header field. The "HTTP2-Settings" header field is a hop-by-hop header field that includes settings that govern the HTTP/2.0 connection, provided in anticipation of the server accepting the request to upgrade. A server MUST reject an attempt to upgrade if this header is not present.

```
HTTP2-Settings    = token68
```

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

The client MUST include values for the following settings (Section 6.5.1):

- o SETTINGS_MAX_CONCURRENT_STREAMS

- o SETTINGS_INITIAL_WINDOW_SIZE

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/2.0.

A server decodes and interprets these values as it would any other SETTINGS frame. Providing these values in the Upgrade request ensures that the protocol does not require default values for the above settings, and gives a client an opportunity to provide other settings prior to receiving any frames from the server.

3.3. 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 3.5).

3.4. 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, after the connection header (Section 3.5). 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.

3.5. Connection Header

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

The client connection header is a sequence of 24 octets, which in hex notation are:

```
505249202a20485454502f322e300d0a0d0a534d0d0a0d0a
```

(the string "PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n") followed by a SETTINGS frame (Section 6.5). 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 6.5) 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 6.8) MAY be omitted if it is clear that the peer is not using HTTP/2.0.

4. HTTP Frames

Once the HTTP/2.0 connection is established, endpoints can begin exchanging frames.

4.1. Frame Header

All frames begin with an 8-octet header followed by a payload of between 0 and 65,535 octets.

```
  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 Payload (0...)                      ...
 +---------------------------------------------------------------+
```

Frame Header

The fields of the frame header are defined as:

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

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

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). 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 frame payload is dependent entirely on the frame type.

4.2. Frame Size

The maximum size of a frame payload varies by frame type and use. The absolute maximum size is 65,535 octets. All implementations SHOULD be capable of receiving and minimally processing frames up to this size.

Certain frame types, such as PING (see Section 6.7), impose additional limits on the amount of payload data allowed. Likewise, additional size limits can be set by specific application uses (see Section 9).

If a frame size exceeds any defined limit, or is too small to contain mandatory frame data, the endpoint MUST send a FRAME_TOO_LARGE error. Frame size errors in frames that affect connection-level state MUST be treated as a connection error (Section 5.4.1).

4.3. Header Compression and Decompression

A header in HTTP/2.0 is a name-value pair with one or more associated values. They are used within HTTP request and response messages as well as server push operations (see Section 8.2).

Header sets are logical collections of zero or more header fields arranged at the application layer. When transmitted over a connection, the 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) or PUSH_PROMISE (Section 6.6) frames. The receiving endpoint reassembles the header block by concatenating the individual fragments, then decompresses the block to reconstruct the header set.

Header block fragments can only be sent as the payload of HEADERS or PUSH_PROMISE frames.

A compressed and encoded header block is transmitted in one or more

HEADERS or PUSH_PROMISE frames. If the number of octets in the block is greater than the space remaining in the frame, the block is divided into multiple fragments, which are then transmitted in multiple frames.

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 frames MUST have the END_HEADERS flag set. The last frame in a sequence of PUSH_PROMISE frames MUST have the END_PUSH_PROMISE flag set.

HEADERS and PUSH_PROMISE frames carry data that can modify the compression context maintained by a receiver. An endpoint receiving HEADERS or PUSH_PROMISE frames MUST reassemble header blocks and perform decompression even if the frames are to be discarded, which is likely to occur after a stream is reset. 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 HEADER and DATA frames exchanged between the client and server within an HTTP/2.0 connection. Streams have several important characteristics:

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

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

- Streams can be closed by either endpoint.

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

- Streams are identified by an integer. Stream identifiers are assigned to streams by the endpoint that initiates a stream.

5.1. Stream States

The lifecycle of a stream is shown in Figure 1.

```
                          +--------+
                    PP    |        |    PP
                 ,--------|  idle  |--------.
                /         |        |         \
               v          +--------+          v
        +----------+          |           +----------+
        |          |          | H         |          |
    ,---| reserved |          |           | reserved |---.
    |   | (local)  |          v           | (remote) |   |
    |   +----------+      +--------+      +----------+   |
    |      |          ES  |        |  ES          |      |
    |      | H    ,-------|  open  |-------.      | H    |
    |      |     /        |        |        \     |      |
    |      v    v         +--------+         v    v      |
    |   +----------+          |           +----------+   |
    |   |   half   |          |           |   half   |   |
    |   |  closed  |          | R         |  closed  |   |
    |   | (remote) |          |           | (local)  |   |
    |   +----------+          |           +----------+   |
    |        |                v                 |        |
    |        |  ES / R    +--------+  ES / R    |        |
    |        `----------->|        |<-----------'        |
    |  R                  | closed |                  R  |
    `-------------------->|        |<--------------------'
                          +--------+
```

Figure 1: Stream States

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

Streams have the following states:

idle:
All streams start in the "idle" state. In this state, no frames have been exchanged.

The following transitions are valid from this state:

* Sending or receiving a HEADERS frame causes the stream to become "open". The stream identifier is selected as described in Section 5.1.1.

* 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 any other type of frame in this state.

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.

Receiving any other type of frame MUST be treated as a stream error (Section 5.4.2) of type PROTOCOL_ERROR.

open:
The "open" state is where both peers can send frames. 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 a END_STREAM flag set, which causes the stream to transition into one of the "half closed" states: an endpoint sending a END_STREAM flag causes the stream state to become "half closed (local)"; an endpoint

receiving a 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 "half closed (local)" cannot be used for sending frames.

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

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 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 a 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 a frame after receiving a RST_STREAM or a frame containing a END_STREAM flag on that stream MUST treat that as a stream error (Section 5.4.2) of type STREAM_CLOSED.

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 that sends a RST_STREAM frame 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.

An endpoint might receive a PUSH_PROMISE frame after it sends RST_STREAM. PUSH_PROMISE causes a stream to become "reserved". If promised streams are not desired, a RST_STREAM can be used to

close any of those streams.

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 message; the stream identifier zero MUST NOT be used to establish a new stream.

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

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

5.1.2. Stream Concurrency

A peer can limit the number of concurrently active streams using the SETTINGS_MAX_CONCURRENT_STREAMS parameters within a SETTINGS frame. The maximum concurrent streams setting is specific to each endpoint and applies only to the peer that receives the setting. That is, clients specify the maximum number of concurrent streams the server can initiate, and servers specify the maximum number of concurrent streams the client can initiate. Endpoints MUST NOT exceed the limit set by their peer.

Streams that are in the "open" state, or either of the "half closed" states count toward the maximum number of streams that an endpoint is permitted to open. Streams in any of these three states count toward the limit advertised in the SETTINGS_MAX_CONCURRENT_STREAMS setting (see Section 6.5.2).

Streams in either of the "reserved" states do not count as open, even if a small amount of application state is retained to ensure that the promised stream can be successfully used.

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/2.0 provides for flow control through use of the WINDOW_UPDATE (Section 6.9) frame type.

5.2.1. 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 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 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 sending a window update frame with a specific flag. See Ending Flow Control (Section 6.9.4) for more details.

7. HTTP/2.0 standardizes only the format of the WINDOW_UPDATE frame (Section 6.9). This does not stipulate how a receiver decides when to send this frame or the value that it sends. Nor does it specify how a sender chooses to send packets. Implementations are able to select any algorithm that suits their needs.

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

5.2.2. Appropriate Use of Flow Control

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

Deployments that do not require this capability 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.

5.3. 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 a specified priority relative to 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 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.

Unless explicitly specified in the HEADERS frame (Section 6.2) during stream creation, the default stream priority is 2^30. Pushed streams (Section 8.2) are assumed to inherit the priority of the associated stream plus one (or 2^31-1 if the the associated stream priority is 2^31-1), i.e. they have priority one lower than the associated stream.

5.4. 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.

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 (Section 6.8) 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.

5.4.2. Stream Error Handling

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

An endpoint that detects a stream error sends a RST_STREAM (Section 6.4) 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 4.3)).

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

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

5.4.3. Connection Termination

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

6. Frame Definitions

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

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

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.

The DATA frame defines the following flags:

END_STREAM (0x1): Bit 1 being set indicates that this frame is the last that the endpoint will send for the identified stream. Setting this flag causes the stream to enter a "half closed" state (Section 5.1).

RESERVED (0x2): Bit 2 is reserved for future use.

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

6.2. HEADERS

The HEADERS frame (type=0x1) carries name-value pairs. The HEADERS is used to open a stream (Section 5.1). Any number of HEADERS frames can be sent on an existing stream at any 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 Fragment (*)                 ...
+---------------------------------------------------------------+
```

HEADERS Frame Payload

The HEADERS frame defines the following flags:

END_STREAM (0x1): Bit 1 being set indicates that this frame is the last that the endpoint will send for the identified stream. Setting this flag causes the stream to enter a "half closed" state (Section 5.1).

RESERVED (0x2): Bit 2 is reserved for future use.

END_HEADERS (0x4): The END_HEADERS bit indicates that this frame ends the sequence of header block fragments necessary to provide a complete set of headers.

The payload for a complete header block is provided by a sequence of HEADERS frames, terminated by a HEADERS frame with the END_HEADERS flag set. Once the sequence terminates, the payload of all HEADERS frames are concatenated and interpreted as a single block.

A HEADERS frame without the END_HEADERS flag set 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 5.4.1) of type PROTOCOL_ERROR.

PRIORITY (0x8): Bit 4 being set indicates that the first four octets of this frame contain a single reserved bit and a 31-bit priority; see Section 5.3. If this bit is not set, the four bytes do not appear and the frame only contains a header block fragment.

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

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 defined in Section 4.3.

6.3. PRIORITY

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

```
 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)                        |
+-+-------------------------------------------------------------+
```

PRIORITY Frame Payload

The payload of a PRIORITY frame contains a single reserved bit and a 31-bit priority.

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.

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 whose stream identifier field is 0x0 the recipient MUST respond with 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. 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 them 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 provided setting value applies.

The SETTINGS frame does not define any flags.

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

The SETTINGS frame affects connection state. A badly formed or incomplete SETTINGS frame MUST be treated as a connection error (Section 5.4.1).

6.5.1. Setting Format

The payload of a SETTINGS frame consists of zero or more settings. Each setting consists of an 8-bit reserved field, 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Reserved (8) |             Setting Identifier (24)           |
+---------------+-----------------------------------------------+
|                        Value (32)                             |
+---------------------------------------------------------------+
```

Setting Format

6.5.2. Defined Settings

The following settings are defined:

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_INITIAL_WINDOW_SIZE (7): indicates the sender's initial window size (in bytes) for stream level flow control.

This settings affects the window size of all streams, including existing streams, see Section 6.9.2.

SETTINGS_FLOW_CONTROL_OPTIONS (10): indicates that streams directed to the sender will not be subject to flow control. The least significant bit (0x1) of the value 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 6.9.4.

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 minimal set of headers that provide additional context for the stream. Section 8.2 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 Fragment (*)                   ...
+---------------------------------------------------------------+
```

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 5.1.1)).

Following the "Promised-Stream-ID" is a header block fragment (Section 4.3).

PUSH_PROMISE frames MUST be associated with an existing, peer-initiated stream. 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.

The PUSH_PROMISE frame defines the following flags:

END_PUSH_PROMISE (0x1): The END_PUSH_PROMISE bit indicates that this frame ends the sequence of header block fragments necessary to provide a complete set of headers.

The payload for a complete header block is provided by a sequence of PUSH_PROMISE frames, terminated by a PUSH_PROMISE frame with the END_PUSH_PROMISE flag set. Once the sequence terminates, the payload of all PUSH_PROMISE frames are concatenated and interpreted as a single block.

A PUSH_PROMISE frame without the END_PUSH_PROMISE flag set MUST be followed by a PUSH_PROMISE 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.

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

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

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

6.7. PING

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

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

PING Payload Format

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

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

The PING frame defines the following flags:

PONG (0x1): Bit 1 being set indicates that this PING frame is a PING response. An endpoint MUST set this flag in PING responses. An endpoint MUST NOT respond to PING frames containing this flag.

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

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

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

After sending a GOAWAY frame, the sender can discard frames for new streams. However, any frames that alter connection state cannot be completely ignored. For instance, HEADERS and PUSH_PROMISE frames MUST be minimally processed to ensure a consistent compression state (see Section 4.3).

```
 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)                          |
+---------------------------------------------------------------+
|                  Additional Debug Data (*)                    |
+---------------------------------------------------------------+
```

GOAWAY Payload Format

The GOAWAY frame does not define any 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 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.

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

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 data MUST NOT be persistently stored, since it could contain sensitive information.

6.9. 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 5.4.2) or connection error (Section 5.4.1) of type FLOW_CONTROL_ERROR if it is unable 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|X|              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 defines the following flags:

END_FLOW_CONTROL (0x1): Bit 1 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.

6.9.1. 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 END_STREAM flag set (for example, an empty data frame) MAY be sent if there is no available

space in either flow control window.

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

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

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

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

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

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

6.9.2. 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 65535 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, an endpoint can only use the default initial window size when sending flow controlled frames. Similarly, the connection flow control window is set to the default initial window size until a WINDOW_UPDATE frame is received.

A SETTINGS frame can alter the initial flow control window size for all current streams. When the value of SETTINGS_INITIAL_WINDOW_SIZE changes, a receiver MUST adjust the size of all stream flow control windows that it maintains by the difference between the new value and the old value. A SETTINGS frame cannot alter the connection flow

control window.

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 client sends 64KB 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 -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.

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.

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.

6.9.4. Ending Flow Control

After a receiver reads in a frame that marks the end of a stream (for example, a data stream with a END_STREAM 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 on the connection using the SETTINGS_FLOW_CONTROL_OPTIONS setting. An implementation that does not wish to perform stream 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.

7. Error Codes

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

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

The following error codes are defined:

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

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

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

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

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 refuses the stream prior to performing any application processing, see Section 8.1.5 for details.

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

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

8. 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.

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 request or response each consist of:

- one contiguous sequence of HEADERS frames;
- zero or more DATA frames; and
- optionally, a contiguous sequence of HEADERS frames

The last frame in the sequence bears an END_STREAM flag.

Other frames, including HEADERS, MAY be interspersed with these frames, but those frames do not carry HTTP semantics.

Trailing header fields are carried in a header block that also terminates the stream. That is, a sequence of HEADERS frames that carries an END_STREAM flag on the last frame. 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, which places the stream in the "closed" state.

8.1.1. Examples

For example, an HTTP GET request that includes request header fields and no body, is transmitted as a single contiguous sequence of HEADERS frames containing the serialized block of request header fields. The last HEADERS frame in the sequence has both the END_HEADERS and END_STREAM flag set:

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

Similarly, a response that includes only response header fields is transmitted as a sequence of HEADERS frames containing the serialized block of response header fields. The last HEADERS frame in the sequence has both the END_HEADERS and END_STREAM flag set:

```
  HTTP/1.1 204 No Content       HEADERS
  Content-Length: 0        ===>   + END_STREAM
                                  + END_HEADERS
                                    :status = 204
                                    content-length: 0
```

An HTTP POST request that includes request header fields and payload data is transmitted as one or more HEADERS frames containing the request headers followed by one or more DATA frames, with the last 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
                                     :scheme = https
  {binary data}                      :host = example.org
                                     :path = /resource
                                     content-type = image/jpeg
                                     content-length = 123

                                 DATA
                                   + END_STREAM
                                     {binary data}
```

A response that includes header fields and payload data is transmitted as one or more HEADERS frames followed by one or more DATA frames, with the last DATA frame in the sequence having the END_STREAM flag set:

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

                                  DATA
                                    + END_STREAM
                                      {binary data}
```

Trailing header fields are sent as a header block after both the request or response header block and all the DATA frames have been sent. The sequence of HEADERS frames that bears the trailers includes a terminal frame that has both END_HEADERS and END_STREAM flags set.

```
  HTTP/1.1 200 OK                HEADERS
  Content-Type: image/jpeg ===>    - END_STREAM
  Content-Length: 123              + END_HEADERS
  TE: trailers                       :status        = 200
  123                                content-type   = image/jpeg
  {binary data}                      content-length = 123
  0
  Foo: bar                       DATA
                                   - END_STREAM
                                     {binary data}

                                 HEADERS
                                   + END_STREAM
                                   + END_HEADERS
                                     foo: bar
```

8.1.2. Request Header Fields

The definitions of the request header fields 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.

- o The host and optional port portions of the request URI (see [RFC3986], Section 3.2), are specified using the new :host header field. [[anchor13: Ed. Note: it needs to be clarified whether or not this replaces the existing HTTP/1.1 Host header.]]

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

- o 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.

- o The Connection, Host, Keep-Alive, Proxy-Connection, and Transfer-Encoding header fields are no longer valid and MUST NOT be sent. [[anchor14: Ed. Note: And "TE" I presume?]]

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. [[anchor15: Ed. Note: This requirement is currently pending review. Consider it "on hold" for the moment.]]

All HTTP Requests that include a body SHOULD include the "content-length" header field. If a server receives a request where the sum of the DATA frame payload lengths does not equal the value of the "content-length" header field, the server MUST return a 400 (Bad Request) error.

If a client omits a mandatory header field from the request, the server MUST reply with a HTTP 400 Bad Request reply.

8.1.3. Response Header Fields

The definitions of the response header fields are largely unchanged relative to HTTP/1.1, with a few notable exceptions:

- o The response status line has been reduced to a single ":status" header field whose value specifies only the numeric response status code. The status text component of the HTTP/1.1 response has been dropped entirely.

- o The response MUST contain exactly one :status header field with exactly one response status value. If the client receives an HTTP response that does not include the :status field, or provides multiple response status code values, it MUST respond with a stream error (Section 5.4.2) of type PROTOCOL_ERROR.

- o 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.

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

Header fields whose names begin with ":" (whether defined in this document or future extensions to this document) MUST appear before any other header fields. [[anchor16: Ed. Note: This requirement is currently pending review. Consider it "on hold" for the moment.]]

8.1.4. GZip Content-Encoding

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

8.1.5. Request Reliability Mechanisms in HTTP/2.0

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/2.0 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.

In both cases, clients MAY automatically retry all requests, including 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/2.0 enables a server to pre-emptively send (or "push") multiple associated resources 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 resources available in order to fully process the originally requested resource.

Pushing additional resources is optional, and is negotiated only between individual endpoints. For instance, an intermediary could receive pushed resources from the server but is not required to forward those on to the client. How to make use of the pushed resources is up to that intermediary. Equally, the intermediary might choose to push additional resources to the client, without any action taken by the server.

Server push is semantically equivalent to a server responding to a GET request for that resource. The PUSH_PROMISE frame, or frames, sent by the server includes a header block that contains the request headers that the server has assumed.

Pushed resources are always associated with an explicit request from a client. The PUSH_PROMISE frames sent by the server are sent on the stream created for the original request. The PUSH_PROMSE frame includes a promised stream identifier, chosen from the stream identifiers available to the server (see Section 5.1.1). Any header fields that are not specified in the PUSH_PROMISE frames sent by the server are inherited from the original request sent by the client.

The header fields in PUSH_PROMISE MUST include the ":scheme", ":host" and ":path" header fields that identify the resource that is being pushed. A PUSH_PROMISE always implies an HTTP method of GET. If a client receives a PUSH_PROMISE that does not include these header fields, or a value for the ":method" header field, it MUST respond with a stream error (Section 5.4.2) of type PROTOCOL_ERROR.

After sending the PUSH_PROMISE frame, the server can begin delivering the pushed resource on a new, server-initiated stream that uses the promised stream identifier. This stream is already implicitly "half

closed" to the client (Section 5.1). The server uses this stream to transmit an HTTP response, using the same sequence of frames as defined in Section 8.1.

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

The server SHOULD send PUSH_PROMISE (Section 6.6) frames prior to sending any HEADERS or DATA frames that reference the promised resources. This avoids a race where clients issue requests for resources 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 ensure that the client is able to see the promises before discovering the resources. Likewise, if the server pushes resources 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 those resources.

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)" to the server. PUSH_PROMISE frames can be interspersed within the frames that comprise response, with the exception that they cannot be interspersed with HEADERS frames that comprise a single header block.

A client can use the SETTINGS_MAX_CONCURRENT_STREAMS setting to limit the number of resources 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.

The request header fields provided in the PUSH_PROMISE frame SHOULD include enough information for a client to determine whether a cached representation of the resource is already available. If the client determines, for any reason, that it does not wish to receive the pushed resource from the server, or if the server takes too long to begin sending the promised resource, the client can send an RST_STREAM frame, using either the CANCEL or REFUSED_STREAM codes, and referencing the pushed stream's identifier.

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 [[anchor17: Ed: weaselly use of "generally", needs better definition]] not permitted to push a response for "www.example.org".

9. Additional HTTP Requirements/Considerations

TODO: SNI, gzip and deflate Content-Encoding, etc..

9.1. Frame Size Limits for HTTP

Frames used for HTTP messages MUST NOT exceed 2^14-1 (16383) octets in length, not counting the 8 octet frame header. An endpoint MUST treat the receipt of a larger frame as a FRAME_TOO_LARGE error (see Section 4.2).

9.2. Connection Management

HTTP/2.0 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/2.0 connection to a given origin ([RFC6454]) concurrently. A client can create additional connections as replacements, either to replace connections that are near to exhausting the available stream identifiers (Section 5.1.1), or to replace connections that have encountered errors (Section 5.4.1).

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

10. Security Considerations

10.1. 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.

10.2. 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. [[anchor23: Issue: This is no longer true]]

10.3. Cacheability of Pushed Resources

Pushed resources are 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.

11. Privacy Considerations

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.

12. IANA Considerations

This document establishes registries for frame types, error codes and settings. These new registries are entered in a new "Hypertext Transfer Protocol (HTTP) 2.0 Parameters" section.

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

12.1. 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	END_STREAM(1)
1	HEADERS	END_STREAM(1), END_HEADERS(4), PRIORITY(8)
2	PRIORITY	-
3	RST_STREAM	-
4	SETTINGS	-
5	PUSH_PROMISE	END_PUSH_PROMISE(1)
6	PING	PONG(1)
7	GOAWAY	-
9	WINDOW_UPDATE	END_FLOW_CONTROL(1)

Table 1

12.2. 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 7.

12.3. 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 6.5.2.

12.4. 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): RFC XXXX (this document)

Related information: This header field is only used by an HTTP/2.0 client for Upgrade-based negotiation.

13. Acknowledgements

This document includes substantial input from the following individuals:

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

- Gabriel Montenegro and Willy Tarreau (Upgrade mechanism)

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

- Mark Nottingham, Julian Reschke, James Snell, Jeff Pinner (Substantial editorial contributions)

14. References

14.1. Normative References

[COMPRESSION] Ruellan, H. and R. Peon, "HTTP Header Compression", draft-ietf-httpbis-header-compression-00 (work in progress), June 2013.

[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.

[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.

[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.

14.2. Informative References

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

[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-http2-03

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.2. 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.3. 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.4. Since draft-ietf-httpbis-http2-00

Changed title throughout.

Removed section on Incompatibilities with SPDY draft#2.

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

Replaced abstract and introduction.

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

Removed unused references.

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

A.5. 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