HTTPbis Working Group
Internet-Draft
Intended status: Standards Track
Expires: October 5, 2013

M. Belshe
Twist
R. Peon
Google, Inc
M. Thomson, Ed.
Microsoft
A. Melnikov, Ed.
Isode Ltd
April 3, 2013

Hypertext Transfer Protocol version 2.0
draft-ietf-httpbis-http2-02

Abstract

This specification describes an optimised expression of the syntax of the Hypertext Transfer Protocol (HTTP). The HTTP/2.0 encapsulation enables more efficient transfer of representations by providing compressed header fields, simultaneous requests, and also introduces unsolicited push of representations from server to client.

This document is an alternative to, but does not obsolete the HTTP message format. HTTP semantics remain unchanged.

Editorial Note (To be removed by RFC Editor)

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

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

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

Status of This Memo

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

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

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

This Internet-Draft will expire on October 5, 2013.

Copyright Notice

Copyright (c) 2013 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction

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

The HTTP/1.1 encapsulation ensures that only one request can be delivered at a time on a given connection. HTTP/1.1 pipelining, which is not widely deployed, only partially addresses these concerns. Clients that need to make multiple requests therefore use commonly multiple connections to a server or servers in order to reduce the overall latency of those requests. [[anchor1: Need to tune the anti-pipelining comments here.]]

Furthermore, HTTP/1.1 header fields are represented in an inefficient fashion, which, in addition to generating more or larger network packets, can cause the small initial TCP window to fill more quickly than is ideal. This results in excessive latency where multiple requests are made on a new TCP connection.

This document defines an optimized mapping of the HTTP semantics to a TCP connection. This optimization reduces the latency costs of HTTP by allowing parallel requests on the same connection and by using an efficient coding for HTTP header fields. Prioritization of requests lets more important requests complete faster, further improving application performance.

HTTP/2.0 applications have an improved impact on network congestion due to the use of fewer TCP connections to achieve the same effect. Fewer TCP connections compete more fairly with other flows. Long-lived connections are also more able to take better advantage of the available network capacity, rather than operating in the slow start phase of TCP.

The HTTP/2.0 encapsulation also enables more efficient processing of messages by providing efficient message framing. Processing of header fields in HTTP/2.0 messages is more efficient (for entities that process many messages).

1.1. Document Organization

The HTTP/2.0 Specification is split into three parts: starting HTTP/2.0 (Section 2), which covers how a HTTP/2.0 is started; a framing layer (Section 3), which multiplexes a TCP connection into independent, length-prefixed frames; and an HTTP layer (Section 4), which specifies the mechanism for overlaying HTTP request/response

pairs on top of the framing layer. While some of the framing layer concepts are isolated from the HTTP layer, 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/2.0 session.

connection: A transport-level connection between two endpoints.

endpoint: Either the client or server of a connection.

frame: The smallest unit of communication, each containing a frame header.

message: A complete sequence of frames.

receiver: An endpoint that is receiving frames.

sender: An endpoint that is transmitting frames.

server: The endpoint which did not initiate the HTTP/2.0 session.

session: A synonym for a connection.

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

stream: A bi-directional flow of bytes across a virtual channel within a HTTP/2.0 session.

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

2. Starting HTTP/2.0

Just as HTTP/1.1 does, HTTP/2.0 uses the "http:" and "https:" URI schemes. An HTTP/2.0-capable client is therefore required to discover whether a server (or intermediary) supports HTTP/2.0.

Different discovery mechanisms are defined for "http:" and "https:" URIs. Discovery for "http:" URIs is described in Section 2.2; discovery for "https:" URIs is described in Section 2.3.

2.1. HTTP/2.0 Version Identification

HTTP/2.0 is identified using the string "HTTP/2.0". This identification is used in the HTTP/1.1 Upgrade header field, in the TLS-NPN [TLSNPN] [[anchor4: TBD]] 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.

[[anchor5: 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 with the "token" syntax defined in Section 3.2.6 of [HTTP-p1]. Experimenters are encouraged to coordinate their experiments on the ietf-http-wg@w3.org mailing list.

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

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

For example:

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

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

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

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

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

[ HTTP/2.0 session ...
```

Once the server returns the 101 response, both the client and the server send a session header (Section 3.2).

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

A client that makes a request to an "https:" URI without prior knowledge about support for HTTP/2.0 uses TLS [RFC5246] with TLS-NPN [TLSNPN] extension. [[anchor6: TBD, maybe ALPN]]

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

2.4. Starting HTTP/2.0 with Prior Knowledge

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

Prior support for HTTP/2.0 is not a strong signal that a given server will support HTTP/2.0 for future sessions. It is possible for server configurations to change or for configurations to differ between instances in clustered server. Different "transparent" intermediaries - intermediaries that are not explicitly selected by either client or server - are another source of variability.

3. HTTP/2.0 Framing Layer

3.1. Session

The HTTP/2.0 session runs atop TCP ([RFC0793]). The client is the TCP connection initiator.

HTTP/2.0 connections are persistent connections. For best performance, it is expected that clients will not close open connections until the user navigates away from all web pages referencing a connection, or until the server closes the connection. Servers are encouraged to leave connections open for as long as possible, but can terminate idle connections if necessary. When either endpoint closes the transport-level connection, it MUST first send a GOAWAY (Section 3.7.7) frame so that the endpoints can reliably determine if requests finished before the close.

3.2. Session Header

After opening a TCP connection and performing either an HTTP/1.1 Upgrade or TLS handshake, the client sends the client session header. The server replies with a server session header.

The session header provides a final confirmation that both peers agree to use the HTTP/2.0 protocol. The SETTINGS frame ensures that client or server configuration is known as quickly as possible.

The client session header is the 25 byte sequence 0x464f4f202a20485454502f322e300d0a0d0a4241520d0a0d0a (the string "FOO * HTTP/2.0\r\n\r\nBAR\r\n\r\n") followed by a SETTINGS frame (Section 3.7.4). The client sends the client session header immediately after receiving an HTTP/1.1 Upgrade, or after receiving a TLS Finished message from the server.

The client session 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. This doesn't address the concerns raised in [TALKING].

The server session header is a SETTINGS frame (Section 3.7.4). The server sends the server session header immediately after receiving and validating the client session header.

The client sends requests immediately after sending the session header, without waiting to receive a server session header. This ensures that confirming session headers does not add latency.

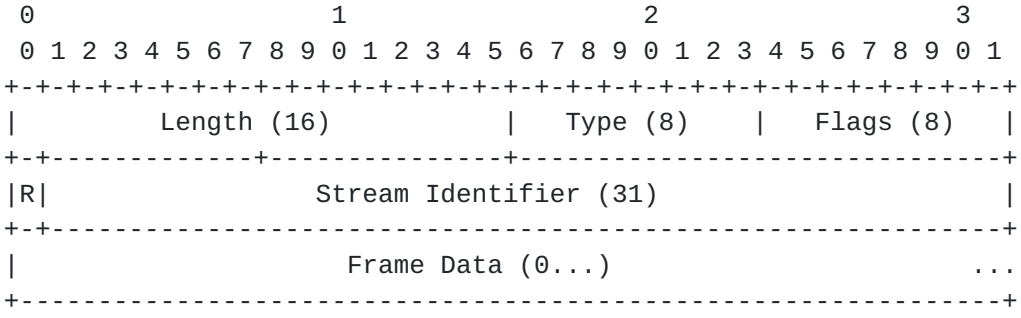
Both client and server MUST close the connection if it does not begin with a valid session header. A GOAWAY frame (Section 3.7.7) MAY be omitted if it is clear that the peer is not using HTTP/2.0.

3.3. Framing

Once the connection is established, clients and servers exchange HTTP/2.0 frames. Frames are the basic unit of communication.

3.3.1. Frame Header

HTTP/2.0 frames share a common header format. Frames have an 8 byte header with between 0 and 65535 bytes of data.



Frame Header

The fields of the frame header are defined as:

Length: The 16-bit length of the frame payload in bytes. The length of the frame header is not included in this sum.

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 frames that use types that they do not support.

Flags: An 8-bit field reserved for flags. Bits that have undefined semantics are reserved. The following flags are defined for all frame types:

FINAL (0x1): Bit 1 (the least significant bit) indicates that this is the last frame in a stream. This places the stream into a half-closed state (Section 3.4.5). No further frames follow in the direction of the carrying frame.

Frame types can define semantics for frame-specific flags.

R: A reserved 1-bit field. The semantics of this bit are not defined.

Stream Identifier: A 31-bit stream identifier (see Section 3.4.1). A value 0 is reserved for frames that are directed at the session as a whole instead of a single stream.

Frame Data: Frames contain between 0 and 65535 bytes of data.

Reserved bits in the frame header MUST be set to zero when sending and MUST be ignored when receiving frames, unless the semantics of the bit are known.

3.3.2. Frame Processing

A frame of the maximum size might be too large for implementations with limited resources to process. Implementations MAY choose to support frames smaller than the maximum possible size. However, implementations MUST be able to receive frames containing at least 8192 octets of payload.

An implementation MUST immediately close a stream if it is unable to process a frame related to that stream due to it exceeding a size limit. The implementation MUST send a RST_STREAM frame (Section 3.7.3) containing FRAME_TOO_LARGE error code if the frame size limit is exceeded.

[[anchor9: <https://github.com/http2/http2-spec/issues/28>: Need a way to signal the maximum frame size; no way to RST_STREAM on non-stream-related frames.]]

3.4. Streams

Streams are independent sequences of bi-directional data divided into frames with several properties:

- Streams can be created by either the client or server.
- Streams optionally carry a set of name-value header pairs.
- Streams can concurrently send data interleaved with other streams.
- Streams can be established and used unilaterally.
- Streams can be cancelled.

3.4.1. Stream Creation

Use of streams does not require negotiation. A stream is not created, streams are used by sending a frame on the stream.

Streams are identified by a 31-bit numeric identifier. Streams initiated by a client use odd numbered stream identifiers. Streams initiated by the server use odd numbered stream identifiers. A stream identifier of zero MUST NOT be used to create a new stream.

The stream identifier of a new stream MUST be greater than all other streams from that endpoint, unless the stream identifier was previously reserved (such as the promised stream identifier in a PUSH_PROMISE (Section 3.7.5) frame). An endpoint that receives an unexpected stream identifier MUST treat this as a session error (Section 3.5.1) of type PROTOCOL_ERROR.

A long-lived session can result in available stream identifiers being exhausted. An endpoint that is unable to create a new stream identifier can establish a new session for any new streams.

An endpoint cannot prevent the creation of a new stream, but it can request the early termination of an unwanted stream. Upon receipt of a frame, the recipient can terminate the corresponding stream by sending a stream error (Section 3.5.2) of type REFUSED_STREAM. This cannot prevent the initiating endpoint from sending frames for that stream prior to receiving this request.

3.4.2. Stream priority

The creator of a stream assigns a priority for that stream. Priority is represented as a 31 bit integer. 0 represents the highest priority and 2^31-1 represents the lowest priority.

The sender and recipient SHOULD use best-effort to process streams in the order of highest priority to lowest priority. [[anchor11: ED: toothless, useless "SHOULD": reword]]

3.4.3. Stream headers

Streams carry optional sets of header fields which carry metadata about the stream. After the stream has been created, and as long as the sender is not closed (Section 3.4.6) or half-closed (Section 3.4.5), each side may send HEADERS frame(s) containing the header data. Header data can be sent in multiple HEADERS frames, and HEADERS frames may be interleaved with data frames.

3.4.4. Stream data exchange

Once a stream is created, it can be used to send arbitrary amounts of data. Generally this means that a series of data frames will be sent on the stream until a frame containing the FINAL flag (Section 3.3.1) is set. Once the FINAL flag has been set on any frame, the stream is considered to be half-closed.

3.4.5. Stream half-close

When one side of the stream sends a frame with the FINAL flag set, the stream is half-closed from that endpoint. The sender of the FINAL flag MUST NOT send further frames on that stream. When both sides have half-closed, the stream is closed.

An endpoint MUST treat the receipt of a data frame on a half-closed stream as a stream error (Section 3.5.2) of type STREAM_CLOSED.

Streams that have never received packets can be considered to be half-closed in the direction that is silent. This allows either peer to create a unidirectional stream, which does not require an explicit close from the peer that does not transmit frames.

3.4.6. Stream close

Streams can be terminated in the following ways:

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

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

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

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

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

If an endpoint receives a data frame after the stream is closed, it MAY send a RST_STREAM to the sender with the status PROTOCOL_ERROR.

3.5. Error Handling

HTTP/2.0 framing permits two classes of error:

- An error condition that renders the entire session unusable is a session error.
- An error in an individual stream is a stream error.

3.5.1. Session Error Handling

A session error is any error which prevents further processing of the framing layer or which corrupts any session state.

An endpoint that encounters a session error MUST first send a GOAWAY (Section 3.7.7) frame with the stream identifier of the last stream that it successfully received from its peer. The GOAWAY frame includes an error code that indicates why the session 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 session error, GOAWAY only provides a best-effort attempt to communicate with the peer about why the session is going down.

An endpoint can end a session at any time. In particular, an endpoint MAY choose to treat a stream error as a session error if the

error is recurrent. Endpoints SHOULD send a GOAWAY frame when ending a session, as long as circumstances permit it.

3.5.2. Stream Error Handling

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

An endpoint that detects a stream error sends a RST_STREAM (Section 3.7.3) frame that contains the stream identifier of the stream where the error occurred. The RST_STREAM frame includes an error code that indicates the type of error.

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

An endpoint SHOULD NOT send more than one RST_STREAM frame for any stream. An endpoint MAY send additional RST_STREAM frames if it receives frames on a closed stream after more than a round trip time. This behaviour is permitted to deal with misbehaving implementations where treating this as a session error is inappropriate.

An endpoint MUST NOT send a RST_STREAM in response to an RST_STREAM frame. This could trigger infinite loops of RST_STREAM frames.

3.5.3. Error Codes

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

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

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

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

INVALID_STREAM (4): A frame was received for an inactive stream.

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

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

REFUSED_STREAM (7): Indicates that the stream was refused before any processing has been done on the stream.

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

3.6. Stream Flow Control

Multiplexing streams introduces contention for access to the shared TCP connection. Stream contention can result in streams being blocked by other streams. A flow control scheme ensures that streams do not destructively interfere with other streams on the same TCP connection.

3.6.1. Flow Control Principles

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

HTTP/2.0 stream flow control aims to allow for future improvements to flow control algorithms without requiring protocol changes. Flow control in HTTP/2.0 has the following characteristics:

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

2. Flow control is based on window update messages. Receivers advertise how many octets they are prepared to receive on a stream. This is a credit-based scheme.

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

4. The initial value for the flow control window is 65536 bytes for both new streams and the overall connection.

5. The frame type determines whether flow control applies to a frame. Of the frames specified in this document, only data frames are subject to flow control; all other frame types do not consume space in the advertised flow control window. This ensures that important control frames are not blocked by flow control.

6. Flow control can be disabled by a receiver. A receiver can choose to either disable flow control for a stream or connection by declaring an infinite flow control limit.

7. HTTP/2.0 standardizes only the format of the window update message (Section 3.7.9). This does not stipulate how a receiver decides when to send this message or the value that it sends. Nor does it specify how a sender chooses to send packets. Implementations are able to select any algorithm that suits their needs.

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

3.6.2. Appropriate Use of Flow Control

Flow control is defined to protect deployments (client, server or intermediary) that are operating under constraints. For example, a proxy must share memory between many connections. Flow control addresses cases where the receiver is unable process data on one stream, yet wants to be continue to process other streams.

Deployments that do not rely on 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. This can lead to suboptimal use of available network resources if flow control is enabled without knowledge of the bandwidth-delay product (see [RFC1323]).

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

3.7. Frame Types

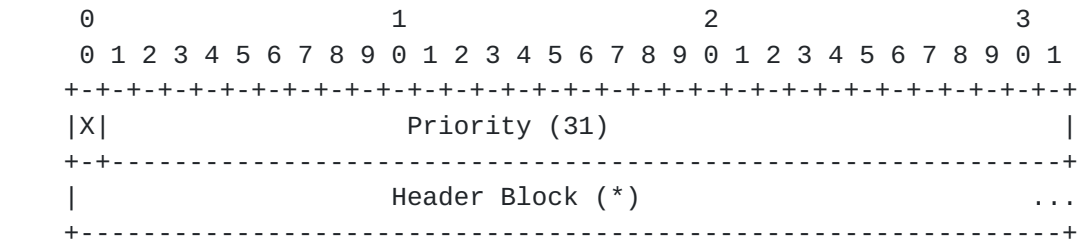
3.7.1. DATA Frames

DATA frames (type=0) are used to convey HTTP message bodies. The payload of a data frame contains either a request or response body.

No frame-specific flags are defined for DATA frames.

3.7.2. HEADERS+PRIORITY

The HEADERS+PRIORITY frame (type=1) allows the sender to set header fields and stream priority at the same time. This MUST be used for each stream that is created.



HEADERS+PRIORITY Frame Payload

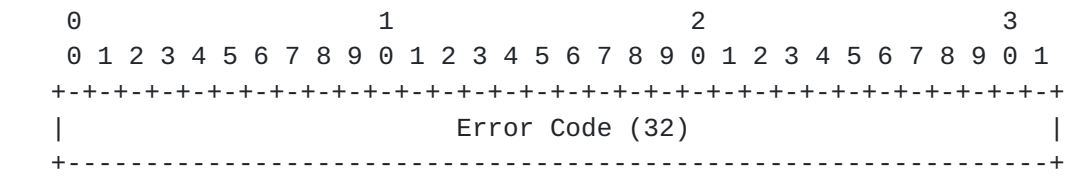
The HEADERS+PRIORITY frame is identical to the HEADERS frame (Section 3.7.8), with a 32-bit field containing priority included before the header block.

The most significant bit of the priority is reserved. The 31-bit priority indicates the priority for the stream, as assigned by the sender, see Section 3.4.2.

3.7.3. RST_STREAM

The RST_STREAM frame (type=3) allows for abnormal termination of a stream. When sent by the creator of a stream, it indicates the

creator wishes to cancel the stream. When sent by the recipient of a stream, it indicates an error or that the recipient did not want to accept the stream, so the stream should be closed.



RST_STREAM Frame Payload

The RST_STREAM frame does not define any valid flags.

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

After receiving a RST_STREAM on a stream, the recipient must not send additional frames for that stream, and the stream moves into the closed state.

3.7.4. SETTINGS

A SETTINGS frame (type=4) contains a set of id/value pairs for communicating configuration data about how the two endpoints may communicate. SETTINGS frames MUST be sent at the start of a session, but they can be sent at any other time by either endpoint. Settings are declarative, not negotiated, each peer indicates their own configuration.

[[anchor17: Note that persistence of settings is under discussion in the WG and might be removed in a future version of this document.]]

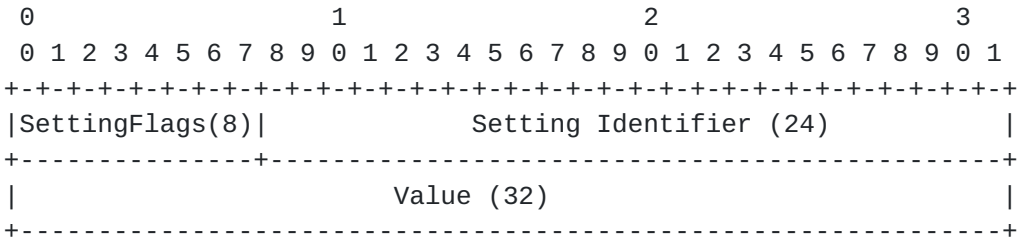
When the server is the sender, the sender can request that configuration data be persisted by the client across HTTP/2.0 sessions and returned to the server in future communications.

Clients persist settings on a per origin basis (see [RFC6454] for a definition of web origins). That is, when a client connects to a server, and the server persists settings within the client, the client SHOULD return the persisted settings on future connections to the same origin AND IP address and TCP port. Clients MUST NOT request servers to use the persistence features of the SETTINGS frames, and servers MUST ignore persistence related flags sent by a client.

Valid frame-specific flags for the SETTINGS frame are:

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

SETTINGS frames always apply to a session, never a single stream. The stream identifier for a settings frame MUST be zero.



SETTINGS ID/Value Pair

The payload of a SETTINGS frame contains zero or more settings. Each setting is comprised of the following

Settings Flags: An 8-bit flags field containing per-setting instructions. The following flags are valid:

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

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

All other settings flags are reserved.

Setting Identifier: A 24-bit field that identifies the setting.

Value: A 32-bit value for the setting.

The following settings are defined:

SETTINGS_UPLOAD_BANDWIDTH (1): allows the sender to send its expected upload bandwidth on this channel. This number is an estimate. The value should be the integral number of kilobytes per second that the sender predicts as an expected maximum upload channel capacity.

SETTINGS_DOWNLOAD_BANDWIDTH (2): allows the sender to send its expected download bandwidth on this channel. This number is an estimate. The value should be the integral number of kilobytes per second that the sender predicts as an expected maximum download channel capacity.

SETTINGS_ROUND_TRIP_TIME (3): allows the sender to send its expected round-trip-time on this channel. The round trip time is defined as the minimum amount of time to send a control frame from this client to the remote and receive a response. The value is represented in milliseconds.

SETTINGS_MAX_CONCURRENT_STREAMS (4): allows the sender to inform the remote endpoint the maximum number of concurrent streams which it 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. For implementers it is recommended that this value be no smaller than 100, so as to not unnecessarily limit parallelism.

SETTINGS_CURRENT_CWND (5): allows the sender to inform the remote endpoint of the current TCP CWND value.

SETTINGS_DOWNLOAD_RETRANS_RATE (6): allows the sender to inform the remote endpoint the retransmission rate (bytes retransmitted / total bytes transmitted).

SETTINGS_INITIAL_WINDOW_SIZE (7): allows the sender to inform the remote endpoint the initial window size (in bytes) for new streams.

SETTINGS_FLOW_CONTROL_OPTIONS (10): This setting allows an endpoint to indicate that streams directed to them will not be subject to flow control. The least significant bit (0x1) is set to indicate that new streams are not flow controlled. Bit 2 (0x2) is set to indicate that the session is not flow controlled. All other bits are reserved.

This setting applies to all streams, including existing streams.

These bits cannot be cleared once set, see Section 3.7.9.4.

The message is intentionally extensible for future information which may improve client-server communications. The sender does not need to send every type of ID/value. It must only send those for which it has accurate values to convey. When multiple ID/value pairs are sent, they should be sent in order of lowest id to highest id. A single SETTINGS frame MUST not contain multiple values for the same

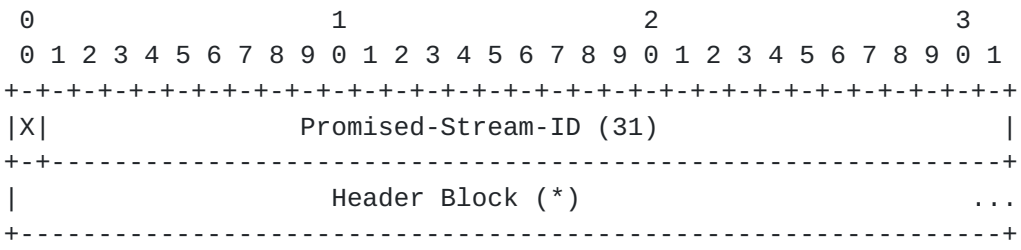
ID. If the recipient of a SETTINGS frame discovers multiple values for the same ID, it MUST ignore all values except the first one.

A server may send multiple SETTINGS frames containing different ID/Value pairs. When the same ID/Value is sent twice, the most recent value overrides any previously sent values. If the server sends IDs 1, 2, and 3 with the FLAG_SETTINGS_PERSIST_VALUE in a first SETTINGS frame, and then sends IDs 4 and 5 with the FLAG_SETTINGS_PERSIST_VALUE, when the client returns the persisted state on its next SETTINGS frame, it SHOULD send all 5 settings (1, 2, 3, 4, and 5 in this example) to the server.

3.7.5. PUSH_PROMISE

The PUSH_PROMISE frame (type=5) allows the sender to signal a promise to create a stream and serve the referenced resource. Minimal data allowing the receiver to understand which resource(s) are to be pushed are to be included.

PUSH_PROMISE frames are sent on an existing stream. They declare the intent to use another stream for the pushing of a resource. The PUSH_PROMISE allows the client an opportunity to reject pushed resources.



PUSH_PROMISE Payload Format

There are no frame-specific flags for the PUSH_PROMISE frame.

The body of a PUSH_PROMISE includes a "Promised-Stream-ID". This 31-bit identifier indicates the stream on which the resource will be pushed. The promised stream identifier MUST be a valid choice for the next stream sent by the sender (see new stream identifier (Section 3.4.1)).

There is no requirement that the streams referred to by this frame are created in the order referenced. The PUSH_PROMISE reserves stream identifiers for later use; these reserved identifiers can be used as prioritization needs dictate.

The PUSH_PROMISE also includes a header block (Section 3.7.10), which

describes the resource that will be pushed.

3.7.6. PING

The PING frame (type=6) is a mechanism for measuring a minimal round-trip time from the sender. PING frames can be sent from the client or the server.

Recipients of a PING frame send an identical frame to the sender as soon as possible. PING should take highest priority if there is other data waiting to be sent.

The PING frame defines a frame-specific flag:

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

The payload of a PING frame contains any value. A PING response MUST contain the contents of the PING request.

3.7.7. GOAWAY

The GOAWAY frame (type=7) informs the remote side of the connection to stop creating streams on this session. 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 session. Recipients of a GOAWAY frame MUST NOT open additional streams on the session, although a new session can be established for new streams. The purpose of this message 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.

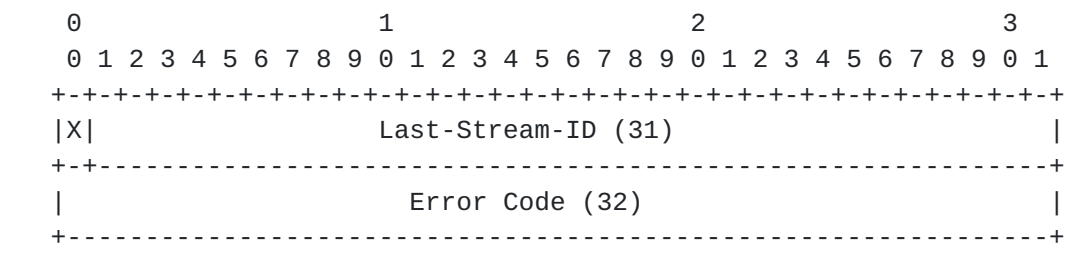
There is an inherent race condition between an endpoint starting new streams and the remote sending a GOAWAY message. To deal with this case, the GOAWAY contains the stream identifier of the last stream which was processed on the sending endpoint in this session. If the receiver of the GOAWAY used streams 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 session).

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

server does not send a GOAWAY frame to indicate where it stopped working).

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

[[anchor18: Issue: session state that is established by those "ignored" messages cannot be ignored without the state in the two peers becoming unsynchronized.]]



GOAWAY Payload Format

The GOAWAY frame does not define any valid flags.

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

The GOAWAY frame contains an identifier of the last stream that the sender of the GOAWAY is prepared to act upon, which can include processing and replies. This allows an endpoint to discover what streams might have had some effect or what might be safe to automatically retry. If no streams were acted upon, the last stream ID MUST be 0.

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

3.7.8. HEADERS

The HEADERS frame (type=8) provides header fields for a stream. It may be optionally sent on an existing stream at any time. Specific application of the headers in this frame is application-dependent.

No frame-specific flags are defined for the HEADERS frame.

The body of a HEADERS frame contains a Headers Block (Section 3.7.10).

3.7.9. WINDOW_UPDATE

The WINDOW_UPDATE frame (type=9) is used to implement flow control in HTTP/2.0.

Flow control in HTTP/2.0 operates at two levels: on each individual stream and on the entire session.

Flow control in HTTP/2.0 is hop by hop, that is, only between the two endpoints of a HTTP/2.0 connection. Intermediaries do not forward WINDOW_UPDATE messages between dependent sessions. However, throttling of data transfer by any recipient 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 frames defined in this document, only data frames are subject to flow control. Receivers MUST either buffer or process all other frames, terminate the corresponding stream, or terminate the session. The stream or session is terminated with a FLOW_CONTROL_ERROR code.

Valid flags for the WINDOW_UPDATE frame are:

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

The WINDOW_UPDATE frame can be stream related or session related. The stream identifier in the WINDOW_UPDATE frame header identifies the affected stream, or includes a value of 0 to indicate that the session flow control window is updated.

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

3.7.9.1. The Flow Control Window

Flow control in HTTP/2.0 is implemented by a flow control window kept by the sender of each stream. The flow control window is a simple integer value that indicates how many bytes of data the sender is permitted to transmit. The flow control window size is a measure of the buffering capability of the recipient.

Two flow control windows apply to the sending of every message: the stream flow control window and the session flow control window. The

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

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

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

The receiver of a message sends a WINDOW_UPDATE frame as it consumes data and frees up space in flow control windows. Separate WINDOW_UPDATE messages are sent for the stream and session 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 session, as appropriate. For streams, the sender sends a RST_STREAM with the error code of FLOW_CONTROL_ERROR code; for the session, a GOAWAY message with a FLOW_CONTROL_ERROR code.

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

3.7.9.2. 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 session flow control window is 65536 bytes. Both endpoints can adjust the initial window size for new streams by including a value for SETTINGS_INITIAL_WINDOW_SIZE in the SETTINGS frame that forms part of the session header.

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

A SETTINGS frame can alter the initial flow control window size for

all current streams. When the value of SETTINGS_INITIAL_WINDOW_SIZE changes, a receiver MUST adjust the size of all flow control windows that it maintains by the difference between the new value and the old value.

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

For example, if the server sets the initial window size to be 16KB, and the client sends 64KB immediately on connection establishment, the client will recalculate the available flow control window to be -48KB on receipt of the SETTINGS frame. The client retains a negative flow control window until WINDOW_UPDATE frames restore the window to being positive, after which the client can resume sending.

3.7.9.3. Reducing the Stream Window Size

A receiver that wishes to use a smaller flow control window than the current size sends 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 messages as it consumes data.

If a receiver decides to accept streams, both sides must recompute the available flow control window based on the initial window size sent in the SETTINGS.

3.7.9.4. Ending Flow Control

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

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

Flow control can be disabled for an individual stream or the overall session 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.

3.7.10. Header Block

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

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

4. HTTP Message Exchanges

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

4.1. Connection Management

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

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

4.1.1. Use of GOAWAY

HTTP/2.0 provides a GOAWAY message which can be used when closing a connection from either the client or server. Without a server GOAWAY message, HTTP has a race condition where the client sends a request just as the server is closing the connection, and the client cannot know if the server received the stream or not. By using the last-stream-id in the GOAWAY, servers can indicate to the client if a request was processed or not.

Note that some servers will choose to send the GOAWAY and immediately terminate the connection without waiting for active streams to finish. The client will be able to determine this because HTTP/2.0 streams are deterministically closed. This abrupt termination will force the client to heuristically decide whether to retry the pending requests. Clients always need to be capable of dealing with this case because they must deal with accidental connection termination cases, which are the same as the server never having sent a GOAWAY.

More sophisticated servers will use GOAWAY to implement a graceful teardown. They will send the GOAWAY and provide some time for the active streams to finish before terminating the connection.

If a HTTP/2.0 client closes the connection, it should also send a GOAWAY message. This allows the server to know if any server-push streams were received by the client.

If the endpoint closing the connection has not received frames on any stream, the GOAWAY will contain a last-stream-id of 0.

4.2. HTTP Request/Response

4.2.1. HTTP Header Fields and HTTP/2.0 Headers

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

4.2.2. Request

The client initiates a request by sending a HEADERS+PRIORITY frame. Requests that do not contain a body MUST set the FINAL flag, indicating that the client intends to send no further data on this stream, unless the server intends to push resources (see

Section 4.3). HEADERS+PRIORITY frame does not contain the FINAL flag for requests that contain a body. The body of a request follows as a series of DATA frames. The last DATA frame sets the FINAL flag to indicate the end of the body.

The header fields included in the HEADERS+PRIORITY frame contain all of the HTTP header fields that are associated with an HTTP request. The header block in HTTP/2.0 is mostly unchanged from today's HTTP header block, with the following differences:

The following fields that are carried in the request line in HTTP/1.1 ([HTTP-p1], Section 3.1.1) are defined as special-valued name-value pairs:

":method": the HTTP method for this request (e.g. "GET", "POST", "HEAD", etc) ([HTTP-p2], Section 4)

":path": ":path" - the request-target for this URI with "/" prefixed (see [HTTP-p1], Section 3.1.1). For example, for "http://www.google.com/search?q=dogs" the path would be "/search?q=dogs". [[anchor26: what forms of the HTTPbis request-target are allowed here?]]

These header fields MUST be present in HTTP requests.

In addition, the following two name-value pairs MUST be present in every request:

":host": the host and optional port portions (see [RFC3986], Section 3.2) of the URI for this request (e.g. "www.google.com:1234"). This header field is the same as the HTTP 'Host' header field ([HTTP-p1], Section 5.4).

":scheme": the scheme portion of the URI for this request (e.g. "https")

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

Header field names MUST be all lowercase.

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

User-agents MUST support gzip compression. Regardless of the Accept-Encoding sent by the user-agent, the server may always send content encoded with gzip or deflate encoding. [[anchor27: Still

valid?]]

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

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

The user-agent is free to prioritize requests as it sees fit. If the user-agent cannot make progress without receiving a resource, it should attempt to raise the priority of that resource. Resources such as images, SHOULD generally use the lowest priority.

If a client sends a HEADERS+PRIORITY frame that omits a mandatory header, the server MUST reply with a HTTP 400 Bad Request reply. [[anchor28: Ed: why PROTOCOL_ERROR on missing ":status" in the response, but HTTP 400 here?]]

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

4.2.3. Response

The server responds to a client request with a HEADERS frame. Symmetric to the client's upload stream, server will send any response body in a series of DATA frames. The last data frame will contain the FINAL flag to indicate the end of the stream and the end of the response. A response that contains no body (such as a 204 or 304 response) consists only of a HEADERS frame that contains the FINAL flag to indicate no further data will be sent on the stream.

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

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

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

All header field names MUST be all lowercase.

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

Responses MAY be accompanied by a Content-Length header field for advisory purposes. This allows clients to learn the full size of an entity prior to receiving all the data frames. This can help in, for example, reporting progress.

If a client receives a response where the sum of the data frame payload length does not equal the size of the Content-Length header field, the client MUST ignore the content length header field. [[anchor29: Ed: See <https://github.com/http2/http2-spec/issues/46>.]]

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

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

4.3. Server Push Transactions

HTTP/2.0 enables a server to send multiple replies to a client for a single request. The rationale for this feature is that sometimes a server knows that it will need to send multiple resources in response to a single request. Without server push features, the client must first download the primary resource, then discover the secondary resource(s), and request them. Pushing of resources avoids the round-trip delay, but also creates a potential race where a server can be pushing content which a user-agent is in the process of requesting. The following mechanics attempt to prevent the race condition while enabling the performance benefit.

Server push is an optional feature. Server push can be disabled by clients that do not wish to receive pushed resources by advertising a SETTINGS_MAX_CONCURRENT_STREAMS SETTING (Section 3.7.4) of zero. This prevents servers from creating the streams necessary to push resources.

Browsers 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 [[anchor30: Ed: weaselly use of "generally", needs better definition]] not permitted to push a

response for "www.example.org".

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

Pushed responses are associated with a request at the HTTP/2.0 framing layer. The PUSH_PROMISE includes a stream identifier for an associated request/response exchange that supplies request header fields. The pushed stream inherits all of the request header fields from the associated stream with the exception of resource identification header fields (":host", ":scheme", and ":path"), which are provided as part of the PUSH_PROMISE frame. Pushed resources always have an associated ":method" of "GET". A cache MUST store these inherited and implied request header fields with the cached resource.

Implementation note: With server push, it is theoretically possible for servers to push unreasonable amounts of content or resources to the user-agent. Browsers MUST implement throttles to protect against unreasonable push attacks. [[anchor31: Ed: insufficiently specified to implement; would like to remove]]

4.3.1. Server implementation

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

A server can push multiple resources in response to a request, but these can only be sent while the response stream remains open. A server MUST NOT send a PUSH_PROMISE on a half-closed stream.

The server SHOULD include any header fields in a PUSH_PROMISE that would allow a cache to determine if the resource is already cached (see [HTTP-p6], Section 4).

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

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

SHOULD be sent prior to any content in the response representation that might allow a client to discover the pushed resource and request it.

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

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

4.3.2. Client implementation

When fetching a resource the client has 3 possibilities:

1. the resource is not being pushed

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

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

When a HEADERS+PRIORITY frame that contains an Associated-To-Stream-ID is received, the client MUST NOT[[anchor34: SHOULD NOT?]] issue GET requests for the resource in the pushed stream, and instead wait for the pushed stream to arrive.

A server MUST NOT push a resource with an Associated-To-Stream-ID of 0. Clients MUST treat this as a session error (Section 3.5.1) of type PROTOCOL_ERROR.

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

To cancel individual server push streams, the client can issue a stream error (Section 3.5.2) of type CANCEL. Upon receipt, the server ceases transmission of the pushed data.

To cancel all server push streams related to a request, the client may issue a stream error (Section 3.5.2) of type CANCEL on the associated-stream-id. By cancelling that stream, the server MUST immediately stop sending frames for any streams with in-association-to for the original stream. [[anchor35: Ed: Triggering side-effects on stream reset is going to be problematic for the framing layer. Purely from a design perspective, it's a layering violation. More practically speaking, the base request stream might already be removed. Special handling logic would be required.]]

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

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

5. Design Rationale and Notes

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

5.1. Separation of Framing Layer and Application Layer

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

This blending is intentional - the primary goal of this protocol is to create a low-latency protocol for use with HTTP. Isolating the two layers is convenient for description of the protocol and how it relates to existing HTTP implementations. However, the ability to reuse the HTTP/2.0 framing layer is a non goal.

5.2. Error handling - Framing Layer

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

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

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

5.3. One Connection Per Domain

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

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

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

5.4. Fixed vs Variable Length Fields

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

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

5.5. Server Push

A subtle but important point is that server push streams must be declared before the associated stream is closed. The reason for this is so that proxies have a lifetime for which they can discard information about previous streams. If a pushed stream could

associate itself with an already-closed stream, then endpoints would not have a specific lifecycle for when they could disavow knowledge of the streams which went before.

6. Security Considerations

6.1. Use of Same-origin constraints

This specification uses the same-origin policy ([RFC6454], Section 3) in all cases where verification of content is required.

6.2. Cross-Protocol Attacks

By utilizing 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. [[anchor45: Issue: This is no longer true]]

6.3. Cacheability of Pushed Resources

Pushed resources do not have an associated request. In order for existing HTTP cache control validations (such as the Vary header field) to work, all cached resources must have a set of request header fields. For this reason, caches MUST be careful to inherit request header fields from the associated stream for the push. This includes the Cookie header field.

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

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

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

7. Privacy Considerations

7.1. Long Lived Connections

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

7.2. SETTINGS frame

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

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

Clients MUST clear persisted SETTINGS information when clearing the cookies.

8. IANA Considerations

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

8.1. Frame Type Registry

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

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

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

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

Table 1

8.2. Error Code Registry

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

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

New registrations are advised to provide the following information:

Error Code: The 32-bit error code value.

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

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

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

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

8.3. Settings Registry

This document establishes a registry for HTTP/2.0 settings. The "HTTP/2.0 Settings" registry manages a 24-bit space. The "HTTP/2.0

Settings" registry operates under the "Expert Review" policy [RFC5226].

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

New registrations are advised to provide the following information:

Setting: The 24-bit setting value.

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

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

Description: A description of the setting. This might include the range of values, any applicable units and how to act upon a value when it is provided.

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

An initial set of settings registrations can be found in Section 3.7.4.

9. 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 and Julian Reschke

10. References

10.1. Normative References

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

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

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

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

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

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

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

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

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

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

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

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

[TLSNPN] Langley, A., "Transport Layer Security (TLS) Next Protocol Negotiation Extension", draft-agl-tls-nextprotoneg-04 (work in progress), May 2012.

10.2. Informative References

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

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

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

A.1. Since draft-ietf-httpbis-http2-01

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

Changed title throughout.

Removed section on Incompatibilities with SPDY draft#2.

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

Replaced abstract and introduction.

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

Removed unused references.

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

A.3. 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 94043
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