HTTPbis Working Group
Internet-Draft
Expires: July 26, 2013

M. Belshe
Twist
R. Peon
Google, Inc
M. Thomson, Ed.
Microsoft
A. Melnikov, Ed.
Isode Ltd
January 22, 2013

Hypertext Transfer Protocol version 2.0
draft-ietf-httpbis-http2-01

Abstract

This document describes an optimised expression of the semantics of the HTTP protocol. The HTTP/2.0 encapsulation enables more efficient transfer of resources over HTTP by providing compressed headers, simultaneous requests, and unsolicited push of resources from server to client.

This document is an alternative to, but does not obsolete RFC{http-p1}. The HTTP protocol semantics described in RFC{http-p2..p7} are unmodified.

Editorial Note (To be removed by RFC Editor)

This draft is a work-in-progress, and does not yet reflect Working Group consensus.

This draft contains features from the SPDY Protocol as a starting point, as per the Working Group's charter. Future drafts will add, remove and change text, based upon the Working Group's decisions.

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/>.

The current issues list is at <http://tools.ietf.org/wg/httpbis/trac/report/21> and related documents (including fancy diffs) can be found at <http://tools.ietf.org/wg/httpbis/>.

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 July 26, 2013.

Copyright Notice

Table of Contents

1. Introduction

HTTP is a wildly successful protocol. HTTP/1.1 message encapsulation [HTTP-p1] 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.

Furthermore, HTTP/1.1 headers 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 headers. 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 headers 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. Definitions

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: A header-prefixed sequence of bytes sent over a HTTP/2.0 session.

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 in using the string "HTTP/2.0". This identification is used in the HTTP/1.1 Upgrade header, in the TLS-NPN [TLSNPN] [[TBD]] field and other places where protocol identification is required.

[[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 corresponding draft number to the identifier before the separator

('/'). For example, draft-ietf-httpbis-http2-03 is identified using the string "HTTP-03/2.0".

Non-compatible experiments that are based on these draft versions MUST include a further identifier. For example, an experimental implementation of packet mood-based encoding based on draft-ietf-httpbis-http2-07 might identify itself as "HTTP-07-emo/2.0". Note that any label MUST conform with the "token" syntax defined in Section 3.2.4 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 [HTTP-p2]. 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 frames ...
```

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. [[Open Issue: This is not definite. We may yet choose to perform negotiation for every connection. Reasons include intermediaries; phased upgrade of load-balanced server farms; etc...]] [[Open Issue: We need to enumerate the ways that clients can learn of HTTP/2.0 support.]]

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

[[TBD, maybe NPN]]

3. HTTP/2.0 Framing Layer

3.1. Session (Connections)

The HTTP/2.0 framing layer (or "session") runs atop a reliable transport layer such as 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 3.6.6) frame so that the endpoints can reliably determine if requests finished before the close.

3.2. Framing

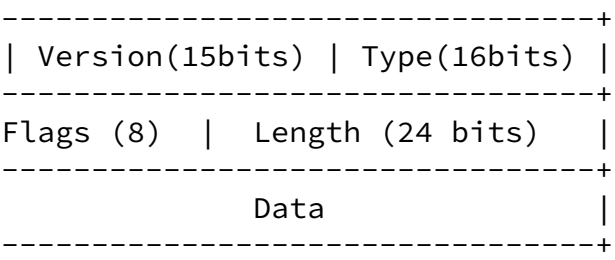
Once the connection is established, clients and servers exchange framed messages. There are two types of frames: control frames (Section 3.2.1) and data frames (Section 3.2.2). Frames always have a common header which is 8 bytes in length.

The first bit is a control bit indicating whether a frame is a control frame or data frame. Control frames carry a version number, a frame type, flags, and a length. Data frames contain the stream ID, flags, and the length for the payload carried after the common header. The simple header is designed to make reading and writing of

frames easy.

All integer values, including length, version, and type, are in network byte order. HTTP/2.0 does not enforce alignment of types in dynamically sized frames.

3.2.1. Control frames



Control bit: The 'C' bit is a single bit indicating if this is a control message. For control frames this value is always 1.

Version: The version number of the HTTP/2.0 protocol. This document describes HTTP/2.0 version 3.

Type: The type of control frame. See Control Frames for the complete list of control frames.

Flags: Flags related to this frame. Flags for control frames and data frames are different.

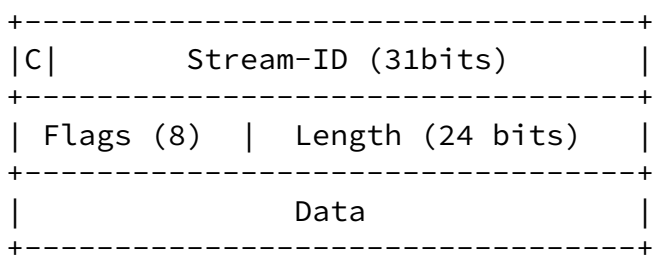
Length: An unsigned 24-bit value representing the number of bytes after the length field.

Data: data associated with this control frame. The format and length of this data is controlled by the control frame type.

Control frame processing requirements:

> Note that full length control frames (16MB) can be large for implementations running on resource-limited hardware. In such cases, implementations MAY limit the maximum length frame supported. However, all implementations MUST be able to receive control frames of at least 8192 octets in length.

3.2.2. Data frames



Control bit: For data frames this value is always 0.

Stream-ID: A 31-bit value identifying the stream.

Flags: Flags related to this frame. Valid flags are:

0x01 = FLAG_FIN - signifies that this frame represents the last frame to be transmitted on this stream. See Stream Close (Section 3.3.7) below.

0x02 = FLAG_COMPRESS - indicates that the data in this frame has been compressed.

Length: An unsigned 24-bit value representing the number of bytes after the length field. The total size of a data frame is 8 bytes + length. It is valid to have a zero-length data frame.

Data: The variable-length data payload; the length was defined in the length field.

Data frame processing requirements:

If an endpoint receives a data frame for a stream-id which is not open and the endpoint has not sent a GOAWAY (Section 3.6.6) frame, it MUST send issue a stream error (Section 3.4.2) with the error code INVALID_STREAM for the stream-id.

If the endpoint which created the stream receives a data frame before receiving a SYN_REPLY on that stream, it is a protocol

error, and the recipient MUST issue a stream error (Section 3.4.2) with the status code PROTOCOL_ERROR for the stream-id.

Implementors note: If an endpoint receives multiple data frames for invalid stream-ids, it MAY close the session.

All HTTP/2.0 endpoints MUST accept compressed data frames. Compression of data frames is always done using zlib compression. Each stream initializes and uses its own compression context

dedicated to use within that stream. Endpoints are encouraged to use application level compression rather than HTTP/2.0 stream level compression.

Each HTTP/2.0 stream sending compressed frames creates its own zlib context for that stream, and these compression contexts MUST be distinct from the compression contexts used with SYN_STREAM/ SYN_REPLY/HEADER compression. (Thus, if both endpoints of a stream are compressing data on the stream, there will be two zlib contexts, one for sending and one for receiving).

3.3. Streams

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

Streams may 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 may be cancelled.

3.3.1. Stream frames

HTTP/2.0 defines 3 control frames to manage the lifecycle of a stream:

SYN_STREAM - Open a new stream

SYN_REPLY - Remote acknowledgement of a new, open stream

RST_STREAM - Close a stream

3.3.2. Stream creation

A stream is created by sending a control frame with the type set to SYN_STREAM (Section 3.6.1). If the server is initiating the stream, the Stream-ID must be even. If the client is initiating the stream, the Stream-ID must be odd. 0 is not a valid Stream-ID. Stream-IDs from each side of the connection must increase monotonically as new streams are created. E.g. Stream 2 may be created after stream 3, but stream 7 must not be created after stream 9. Stream IDs do not wrap: when a client or server cannot create a new stream id without exceeding a 31 bit value, it MUST NOT create a new stream.

The stream-id MUST increase with each new stream. If an endpoint

receives a SYN_STREAM with a stream id which is less than any previously received SYN_STREAM, it MUST issue a session error (Section 3.4.1) with the status PROTOCOL_ERROR.

It is a protocol error to send two SYN_STREAMs with the same stream-id. If a recipient receives a second SYN_STREAM for the same stream, it MUST issue a stream error (Section 3.4.2) with the status code PROTOCOL_ERROR.

Upon receipt of a SYN_STREAM, the recipient can reject the stream by sending a stream error (Section 3.4.2) with the error code REFUSED_STREAM. Note, however, that the creating endpoint may have already sent additional frames for that stream which cannot be immediately stopped.

Once the stream is created, the creator may immediately send HEADERS or DATA frames for that stream, without needing to wait for the recipient to acknowledge.

3.3.2.1. Unidirectional streams

When an endpoint creates a stream with the FLAG_UNIDIRECTIONAL flag set, it creates a unidirectional stream which the creating endpoint can use to send frames, but the receiving endpoint cannot. The receiving endpoint is implicitly already in the half-closed

(Section 3.3.6) state.

3.3.2.2. Bidirectional streams

SYN_STREAM frames which do not use the FLAG_UNIDIRECTIONAL flag are bidirectional streams. Both endpoints can send data on a bi-directional stream.

3.3.3. Stream priority

The creator of a stream assigns a priority for that stream. Priority is represented as an integer from 0 to 7. 0 represents the highest priority and 7 represents the lowest priority.

The sender and recipient SHOULD use best-effort to process streams in the order of highest priority to lowest priority.

3.3.4. Stream headers

Streams carry optional sets of name/value pair headers which carry metadata about the stream. After the stream has been created, and as long as the sender is not closed (Section 3.3.7) or half-closed (Section 3.3.6), 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.

3.3.5. 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 FLAG_FIN flag is set. The FLAG_FIN can be set on a SYN_STREAM (Section 3.6.1), SYN_REPLY (Section 3.6.2), HEADERS (Section 3.6.7) or a DATA (Section 3.2.2) frame. Once the FLAG_FIN has been sent, the stream is considered to be half-closed.

3.3.6. Stream half-close

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

sides have half-closed, the stream is closed.

If an endpoint receives a data frame after the stream is half-closed from the sender (e.g. the endpoint has already received a prior frame for the stream with the FIN flag set), it MUST send a RST_STREAM to the sender with the status STREAM_ALREADY_CLOSED.

3.3.7. Stream close

There are 3 ways that streams can be terminated:

Normal termination: Normal stream termination occurs when both sender and recipient have half-closed the stream by sending a FLAG_FIN.

Abrupt termination: Either the client or server can send a RST_STREAM control frame at any time. A RST_STREAM contains an error code to indicate the reason for failure. When a RST_STREAM is sent from the stream originator, it indicates a failure to complete the stream and that no further data will be sent on the stream. When a RST_STREAM is sent from the stream recipient, the sender, upon receipt, should stop sending any data on the stream. The stream recipient should be aware that there is a race between data already in transit from the sender and the time the RST_STREAM is received. See Stream Error Handling (Section 3.4.2)

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 must send a RST_STREAM to the sender with the status PROTOCOL_ERROR.

3.4. Error Handling

The HTTP/2.0 framing layer has only two types of errors, and they are always handled consistently. Any reference in this specification to "issue a session error" refers to Section 3.4.1. Any reference to "issue a stream error" refers to Section 3.4.2.

3.4.1. Session Error Handling

A session error is any error which prevents further processing of the framing layer or which corrupts the session compression state. When a session error occurs, the endpoint encountering the error MUST first send a GOAWAY (Section 3.6.6) frame with the stream id of most recently received stream from the remote endpoint, and the error code for why the session is terminating. After sending the GOAWAY frame, the endpoint MUST close the TCP connection.

Note that the session compression state is dependent upon both endpoints always processing all compressed data. If an endpoint partially processes a frame containing compressed data without updating compression state properly, future control frames which use compression will be always be errored. Implementations SHOULD always try to process compressed data so that errors which could be handled as stream errors do not become session errors.

Note that because this GOAWAY is sent during a session error case, it is possible that the GOAWAY will not be reliably received by the receiving endpoint. It is a best-effort attempt to communicate with the remote about why the session is going down.

3.4.2. Stream Error Handling

A stream error is an error related to a specific stream-id which does not affect processing of other streams at the framing layer. Upon a stream error, the endpoint MUST send a RST_STREAM (Section 3.6.3) frame which contains the stream id of the stream where the error occurred and the error status which caused the error. After sending the RST_STREAM, the stream is closed to the sending endpoint. After sending the RST_STREAM, if the sender receives any frames other than a RST_STREAM for that stream id, it will result in sending additional RST_STREAM frames. An endpoint MUST NOT send a RST_STREAM in response to an RST_STREAM, as doing so would lead to RST_STREAM loops. Sending a RST_STREAM does not cause the HTTP/2.0 session to be closed.

If an endpoint has multiple RST_STREAM frames to send in succession for the same stream-id and the same error code, it MAY coalesce them into a single RST_STREAM frame. (This can happen if a stream is closed, but the remote sends multiple data frames. There is no

reason to send a RST_STREAM for each frame in succession).

3.5. 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.

3.5.1. 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. The following principles guide the HTTP/2.0 design:

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 [[TBD: ... and for the overall 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. Flow control can be disabled by a receiver. A receiver can choose to either disable flow control, or to declare an infinite flow control limit. [[TBD: determine whether just one mechanism is sufficient, and then which alternative]]

5. HTTP/2.0 standardizes only the format of the window update message (Section 3.6.8). 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. An example flow control algorithm is described in Section 3.5.2.

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.

3.5.2. Basic Flow Control Algorithm

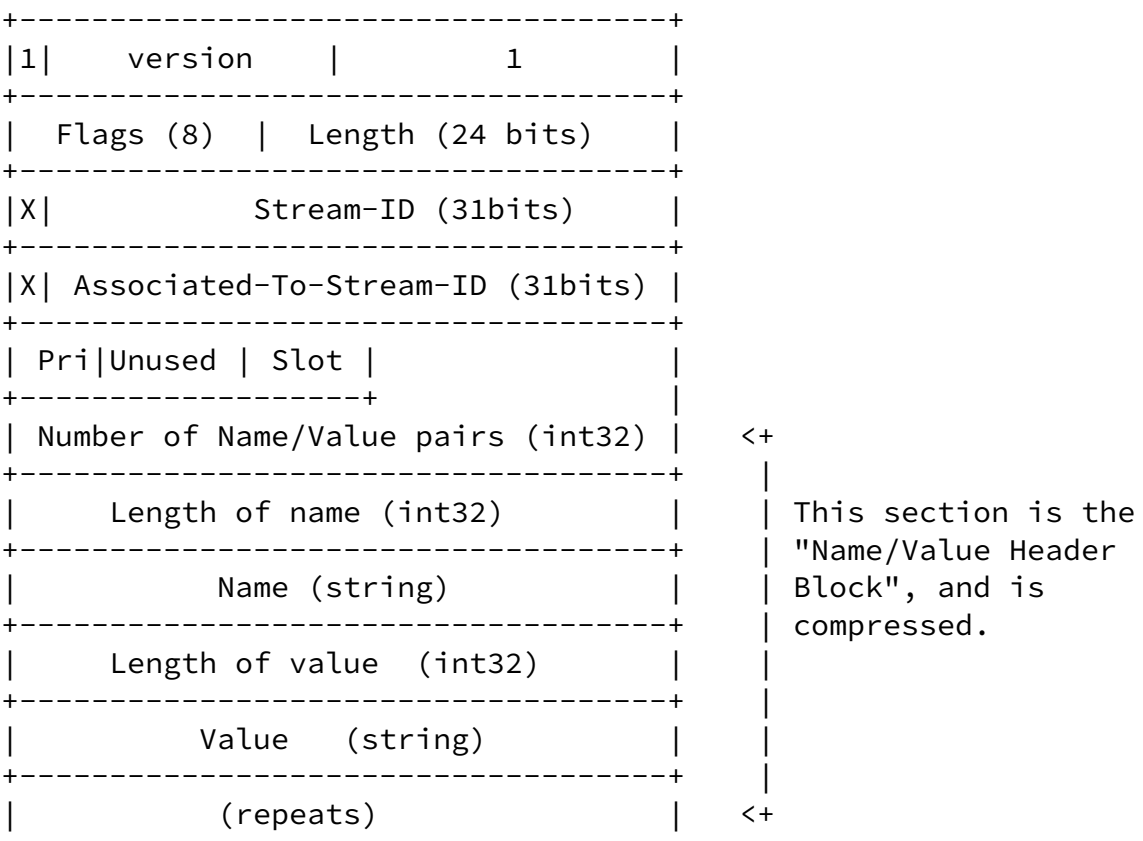
This section describes a basic flow control algorithm. This algorithm is provided as an example, implementations can use any algorithm that complies with flow control requirements.

[[Algorithm TBD]]

3.6. Control frame types

3.6.1. SYN_STREAM

The SYN_STREAM control frame allows the sender to asynchronously create a stream between the endpoints. See Stream Creation (Section 3.3.2)



Flags: Flags related to this frame. Valid flags are:

0x01 = FLAG_FIN - marks this frame as the last frame to be transmitted on this stream and puts the sender in the half-closed (Section 3.3.6) state.

0x02 = FLAG_UNIDIRECTIONAL - a stream created with this flag puts the recipient in the half-closed (Section 3.3.6) state.

Length: The length is the number of bytes which follow the length field in the frame. For SYN_STREAM frames, this is 10 bytes plus the length of the compressed Name/Value block.

Stream-ID: The 31-bit identifier for this stream. This stream-id will be used in frames which are part of this stream.

Associated-To-Stream-ID: The 31-bit identifier for a stream which

this stream is associated to. If this stream is independent of all other streams, it should be 0.

Priority: A 3-bit priority (Section 3.3.3) field.

Unused: 5 bits of unused space, reserved for future use.

Slot: An 8 bit unsigned integer specifying the index in the server's

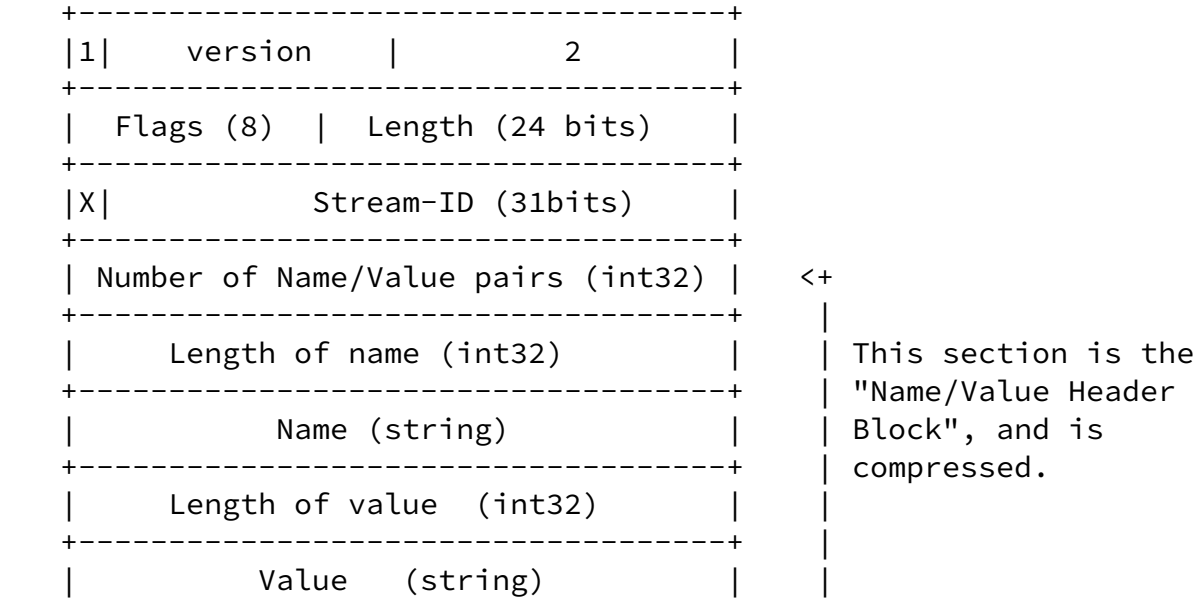
CREDENTIAL vector of the client certificate to be used for this request. see CREDENTIAL frame (Section 3.6.9). The value 0 means no client certificate should be associated with this stream.

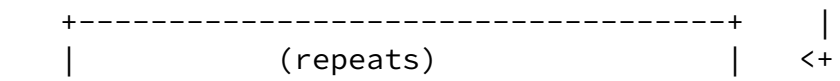
Name/Value Header Block: A set of name/value pairs carried as part of the SYN_STREAM. see Name/Value Header Block (Section 3.6.10).

If an endpoint receives a SYN_STREAM which is larger than the implementation supports, it MAY send a RST_STREAM with error code FRAME_TOO_LARGE. All implementations MUST support the minimum size limits defined in the Control Frames section (Section 3.2.1).

3.6.2. SYN_REPLY

SYN_REPLY indicates the acceptance of a stream creation by the recipient of a SYN_STREAM frame.





Flags: Flags related to this frame. Valid flags are:

0x01 = FLAG_FIN - marks this frame as the last frame to be transmitted on this stream and puts the sender in the half-closed (Section 3.3.6) state.

Length: The length is the number of bytes which follow the length field in the frame. For SYN_REPLY frames, this is 4 bytes plus the length of the compressed Name/Value block.

Stream-ID: The 31-bit identifier for this stream.

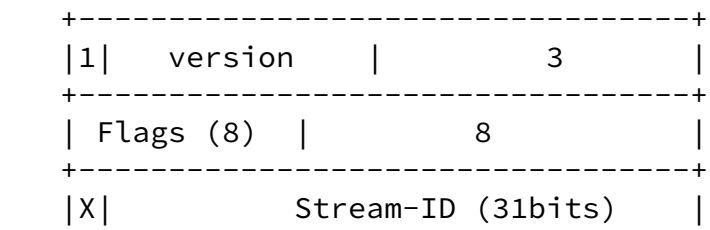
If an endpoint receives multiple SYN_REPLY frames for the same active stream ID, it MUST issue a stream error (Section 3.4.2) with the error code STREAM_IN_USE.

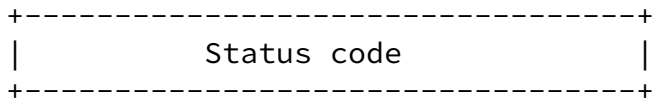
Name/Value Header Block: A set of name/value pairs carried as part of the SYN_STREAM. see Name/Value Header Block (Section 3.6.10).

If an endpoint receives a SYN_REPLY which is larger than the implementation supports, it MAY send a RST_STREAM with error code FRAME_TOO_LARGE. All implementations MUST support the minimum size limits defined in the Control Frames section (Section 3.2.1).

3.6.3. RST_STREAM

The RST_STREAM frame 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.





Flags: Flags related to this frame. RST_STREAM does not define any flags. This value must be 0.

Length: An unsigned 24-bit value representing the number of bytes after the length field. For RST_STREAM control frames, this value is always 8.

Stream-ID: The 31-bit identifier for this stream.

Status code: (32 bits) An indicator for why the stream is being terminated.The following status codes are defined:

1 - PROTOCOL_ERROR. This is a generic error, and should only be used if a more specific error is not available.

2 - INVALID_STREAM. This is returned when a frame is received for a stream which is not active.

3 - REFUSED_STREAM. Indicates that the stream was refused before any processing has been done on the stream.

4 - UNSUPPORTED_VERSION. Indicates that the recipient of a stream does not support the HTTP/2.0 version requested.

5 - CANCEL. Used by the creator of a stream to indicate that the stream is no longer needed.

6 - INTERNAL_ERROR. This is a generic error which can be used when the implementation has internally failed, not due to anything in the protocol.

7 - FLOW_CONTROL_ERROR. The endpoint detected that its peer violated the flow control protocol.

8 - STREAM_IN_USE. The endpoint received a SYN_REPLY for a stream already open.

9 - STREAM_ALREADY_CLOSED. The endpoint received a data or SYN_REPLY frame for a stream which is half closed.

10 - INVALID_CREDENTIALS. The server received a request for a resource whose origin does not have valid credentials in the client certificate vector.

11 - FRAME_TOO_LARGE. The endpoint received a frame which this implementation could not support. If FRAME_TOO_LARGE is sent for a SYN_STREAM, HEADERS, or SYN_REPLY frame without fully processing the compressed portion of those frames, then the compression state will be out-of-sync with the other endpoint. In this case, senders of FRAME_TOO_LARGE MUST close the session.

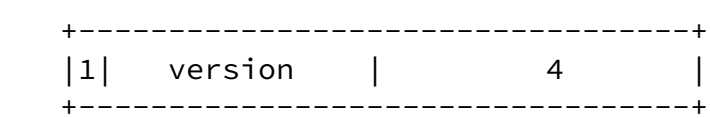
Note: 0 is not a valid status code for a RST_STREAM.

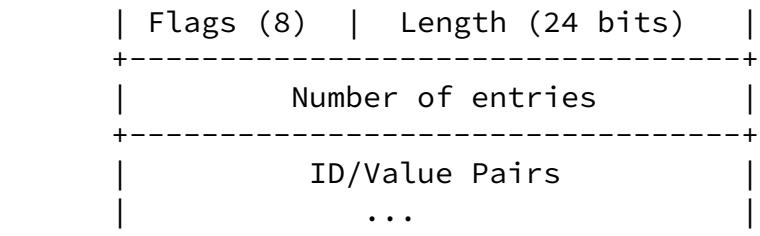
After receiving a RST_STREAM on a stream, the recipient must not send additional frames for that stream, and the stream moves into the closed state.

3.6.4. SETTINGS

A SETTINGS frame contains a set of id/value pairs for communicating configuration data about how the two endpoints may communicate. SETTINGS frames can be sent at any time by either endpoint, are optionally sent, and are fully asynchronous. 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.

Persistence of SETTINGS ID/Value pairs is done on a per origin/IP pair (the "origin" is the set of scheme, host, and port from the URI. See [RFC6454]). 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.





Control bit: The control bit is always 1 for this message.

Version: The HTTP/2.0 version number.

Type: The message type for a SETTINGS message is 4.

Flags: FLAG_SETTINGS_CLEAR_SETTINGS (0x1): When set, the client should clear any previously persisted SETTINGS ID/Value pairs. If this frame contains ID/Value pairs with the FLAG_SETTINGS_PERSIST_VALUE set, then the client will first clear its existing, persisted settings, and then persist the values with the flag set which are contained within this frame. Because persistence is only implemented on the client, this flag can only be used when the sender is the server.

Length: An unsigned 24-bit value representing the number of bytes after the length field. The total size of a SETTINGS frame is 8 bytes + length.

Number of entries: A 32-bit value representing the number of ID/value pairs in this message.

ID: A 32-bit ID number, comprised of 8 bits of flags and 24 bits of unique ID.

ID.flags:

FLAG_SETTINGS_PERSIST_VALUE (0x1): When set, the sender of this SETTINGS frame is requesting that the recipient persist the ID/

Value and return it in future SETTINGS frames sent from the sender to this recipient. Because persistence is only implemented on the client, this flag is only sent by the server.

FLAG_SETTINGS_PERSISTED (0x2): When set, the sender is notifying the recipient that this ID/Value pair was previously sent to the sender by the recipient with the FLAG_SETTINGS_PERSIST_VALUE, and the sender is returning it. Because persistence is only implemented on the client, this flag is only sent by the client.

Defined IDs:

1 - SETTINGS_UPLOAD_BANDWIDTH 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.

2 - SETTINGS_DOWNLOAD_BANDWIDTH 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.

3 - SETTINGS_ROUND_TRIP_TIME 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.

4 - SETTINGS_MAX_CONCURRENT_STREAMS allows the sender to inform the remote endpoint the maximum number of concurrent streams which it will allow. By default there is no limit. For implementors it is recommended that this value be no smaller than 100.

5 - SETTINGS_CURRENT_CWND allows the sender to inform the remote endpoint of the current TCP CWND value.

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

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

8 - SETTINGS_CLIENT_CERTIFICATE_VECTOR_SIZE allows the server to inform the client of the new size of the client certificate vector.

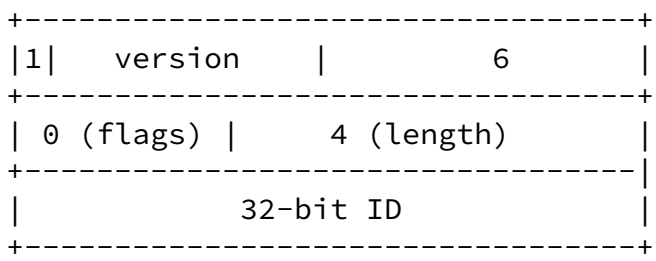
Value: A 32-bit value.

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.

3.6.5. PING

The PING control frame is a mechanism for measuring a minimal round-trip time from the sender. It can be sent from the client or the server. Recipients of a PING frame should send an identical frame to the sender as soon as possible (if there is other pending data waiting to be sent, PING should take highest priority). Each ping sent by a sender should use a unique ID.



Control bit: The control bit is always 1 for this message.

Version: The HTTP/2.0 version number.

Type: The message type for a PING message is 6.

Length: This frame is always 4 bytes long.

ID: A unique ID for this ping, represented as an unsigned 32 bit value. When the client initiates a ping, it must use an odd numbered ID. When the server initiates a ping, it must use an even numbered ping. Use of odd/even IDs is required in order to avoid accidental looping on PINGs (where each side initiates an identical PING at the same time).

Note: If a sender uses all possible PING ids (e.g. has sent all 2^31 possible IDs), it can wrap and start re-using IDs.

If a server receives an even numbered PING which it did not initiate, it must ignore the PING. If a client receives an odd numbered PING which it did not initiate, it must ignore the PING.

3.6.6. GOAWAY

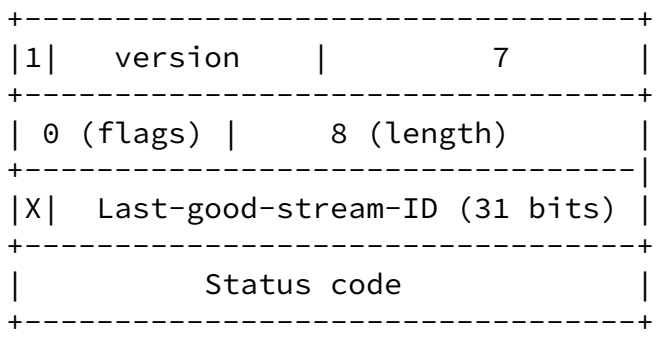
The GOAWAY control frame is a mechanism to tell the remote side of the connection to stop creating streams on this session. It can be sent from the client or the server. Once sent, the sender will not respond to any new SYN_STREAMs on this session. Recipients of a GOAWAY frame must not send additional streams on this 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 sending SYN_STREAMs and the remote sending a GOAWAY message. To deal with this case, the GOAWAY contains a last-stream-id indicating the stream-id of the last stream which was created on the sending endpoint in this session. If the receiver of the GOAWAY sent new SYN_STREAMs for sessions after this last-stream-id, they were not processed by the server and the receiver may treat the stream as though it had never been created at all (hence the receiver may want to re-create the stream 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 must ignore all SYN_STREAM frames for new streams.



Control bit: The control bit is always 1 for this message.

Version: The HTTP/2.0 version number.

Type: The message type for a GOAWAY message is 7.

Length: This frame is always 8 bytes long.

Last-good-stream-Id: The last stream id which was replied to (with either a SYN_REPLY or RST_STREAM) by the sender of the GOAWAY message. If no streams were replied to, this value MUST be 0.

Status: The reason for closing the session.

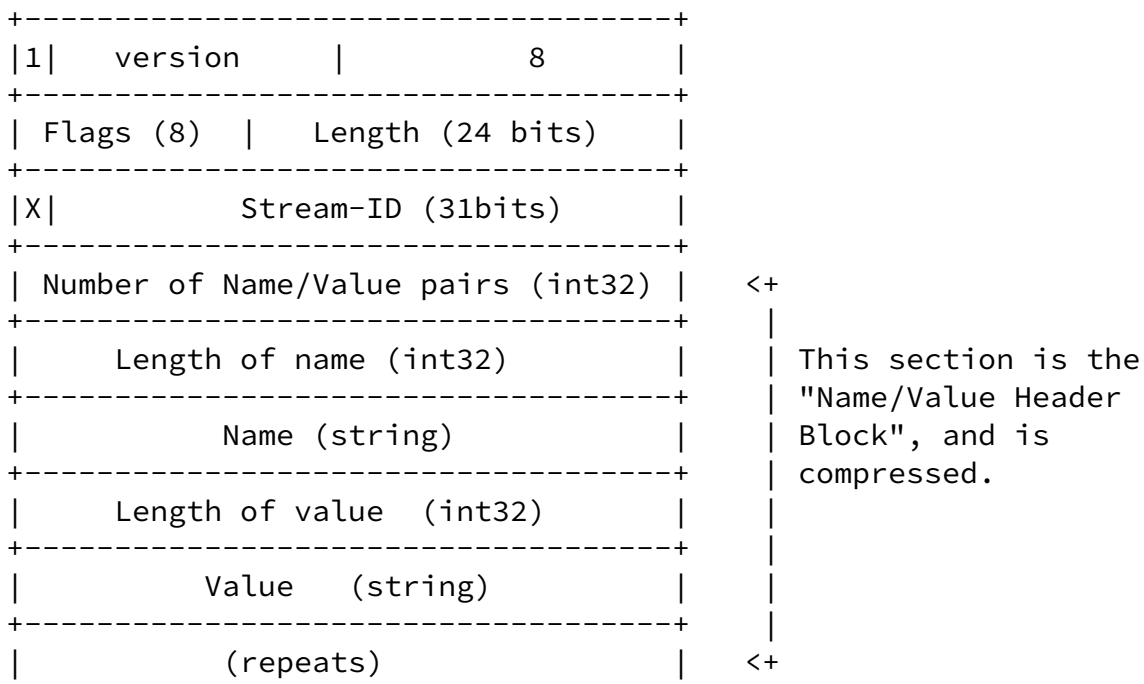
0 - OK. This is a normal session teardown.

1 - PROTOCOL_ERROR. This is a generic error, and should only be used if a more specific error is not available.

2 - INTERNAL_ERROR. This is a generic error which can be used when the implementation has internally failed, not due to anything in the protocol.

3.6.7. HEADERS

The HEADERS frame augments a stream with additional headers. It may be optionally sent on an existing stream at any time. Specific application of the headers in this frame is application-dependent. The name/value header block within this frame is compressed.



Flags: Flags related to this frame. Valid flags are:

0x01 = FLAG_FIN - marks this frame as the last frame to be transmitted on this stream and puts the sender in the half-closed (Section 3.3.6) state.

Length: An unsigned 24 bit value representing the number of bytes after the length field. The minimum length of the length field is 4 (when the number of name value pairs is 0).

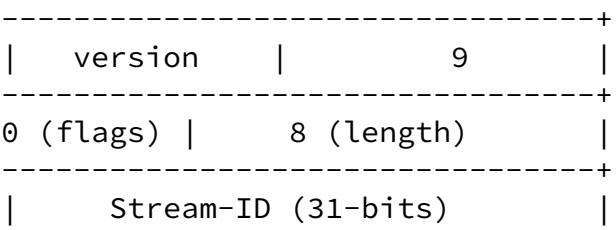
Stream-ID: The stream this HEADERS block is associated with.

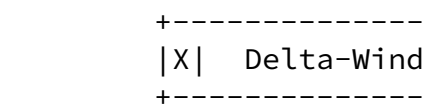
Name/Value Header Block: A set of name/value pairs carried as part of the SYN_STREAM. see Name/Value Header Block (Section 3.6.10).

3.6.8. WINDOW_UPDATE

The WINDOW_UPDATE control frame is used to implement per stream flow control in HTTP/2.0. Flow control in HTTP/2.0 is per hop, that is, only between the two endpoints of a HTTP/2.0 connection. If there are one or more intermediaries between the client and the origin server, flow control signals are not explicitly forwarded by the intermediaries. (However, throttling of data transfer by any recipient may have the effect of indirectly propagating flow control information upstream back to the original sender.) Flow control only applies to the data portion of data frames. Recipients must buffer all control frames. If a recipient fails to buffer an entire control frame, it MUST issue a stream error (Section 3.4.2) with the status code FLOW_CONTROL_ERROR for the stream.

Flow control in HTTP/2.0 is implemented by a data transfer window kept by the sender of each stream. The data transfer window is a simple uint32 that indicates how many bytes of data the sender can transmit. After a stream is created, but before any data frames have been transmitted, the sender begins with the initial window size. This window size is a measure of the buffering capability of the recipient. The sender must not send a data frame with data length greater than the transfer window size. After sending each data frame, the sender decrements its transfer window size by the amount of data transmitted. When the window size becomes less than or equal to 0, the sender must pause transmitting data frames. At the other end of the stream, the recipient sends a WINDOW_UPDATE control back to notify the sender that it has consumed some data and freed up buffer space to receive more data.





Control bit: The control bit is always 1 for this message.

Version: The HTTP/2.0 version number.

Type: The message type for a WINDOW_UPDATE message is 9.

Length: The length field is always 8 for this frame (there are 8 bytes after the length field).

Stream-ID: The stream ID that this WINDOW_UPDATE control frame is for.

Delta-Window-Size: The additional number of bytes that the sender can transmit in addition to existing remaining window size. The legal range for this field is 1 to 2^31 - 1 (0x7fffffff) bytes.

The window size as kept by the sender must never exceed 2^31 (although it can become negative in one special case). If a sender receives a WINDOW_UPDATE that causes the its window size to exceed this limit, it must send RST_STREAM with status code FLOW_CONTROL_ERROR to terminate the stream.

When a HTTP/2.0 connection is first established, the default initial

window size for all streams is 64KB. An endpoint can use the SETTINGS control frame to adjust the initial window size for the connection. That is, its peer can start out using the 64KB default initial window size when sending data frames before receiving the SETTINGS. Because SETTINGS is asynchronous, there may be a race condition if the recipient wants to decrease the initial window size, but its peer immediately sends 64KB on the creation of a new connection, before waiting for the SETTINGS to arrive. This is one case where the window size kept by the sender will become negative. Once the sender detects this condition, it must stop sending data frames and wait for the recipient to catch up. The recipient has two choices:

immediately send RST_STREAM with FLOW_CONTROL_ERROR status code.

allow the head of line blocking (as there is only one stream for the session and the amount of data in flight is bounded by the default initial window size), and send WINDOW_UPDATE as it consumes data.

In the case of option 2, both sides must compute the window size based on the initial window size in the SETTINGS. For example, if the recipient sets the initial window size to be 16KB, and the sender sends 64KB immediately on connection establishment, the sender will discover its window size is -48KB on receipt of the SETTINGS. As the recipient consumes the first 16KB, it must send a WINDOW_UPDATE of 16KB back to the sender. This interaction continues until the sender's window size becomes positive again, and it can resume transmitting data frames.

After the recipient reads in a data frame with FLAG_FIN that marks the end of the data stream, it should not send WINDOW_UPDATE frames as it consumes the last data frame. A sender should ignore all the WINDOW_UPDATE frames associated with the stream after it send the last frame for the stream.

The data frames from the sender and the WINDOW_UPDATE frames from the recipient are completely asynchronous with respect to each other. This property allows a recipient to aggressively update the window size kept by the sender to prevent the stream from stalling.

3.6.9. CREDENTIAL

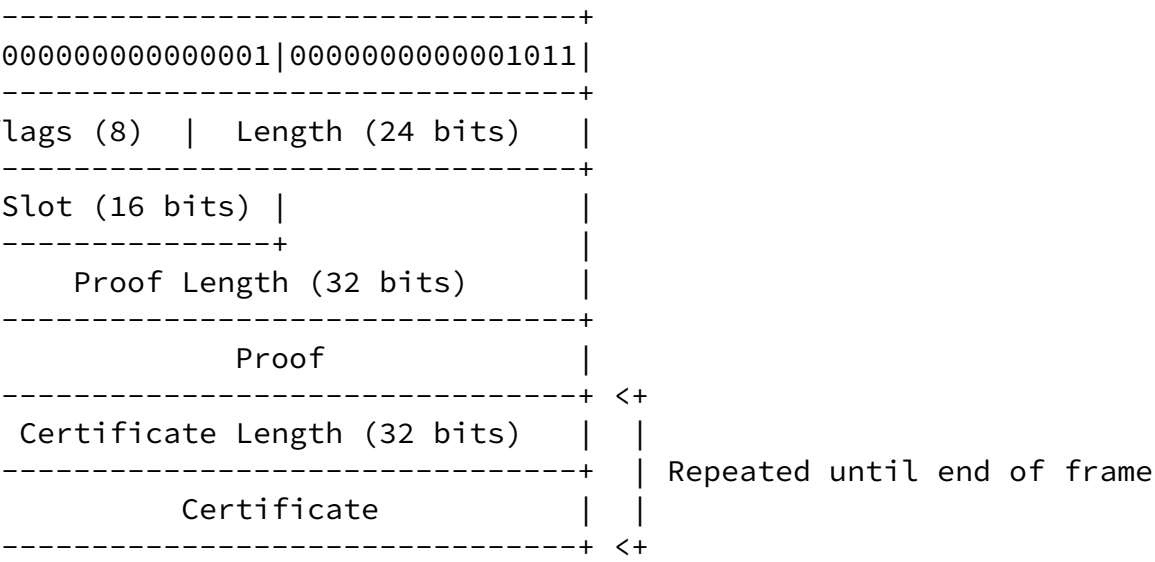
The CREDENTIAL control frame is used by the client to send additional client certificates to the server. A HTTP/2.0 client may decide to send requests for resources from different origins on the same HTTP/2.0 session if it decides that that server handles both origins. For example if the IP address associated with both hostnames matches

and the SSL server certificate presented in the initial handshake is valid for both hostnames. However, because the SSL connection can contain at most one client certificate, the client needs a mechanism to send additional client certificates to the server.

The server is required to maintain a vector of client certificates associated with a HTTP/2.0 session. When the client needs to send a

client certificate to the server, it will send a CREDENTIAL frame that specifies the index of the slot in which to store the certificate as well as proof that the client posesses the corresponding private key. The initial size of this vector must be 8. If the client provides a client certificate during the first TLS handshake, the contents of this certificate must be copied into the first slot (index 1) in the CREDENTIAL vector, though it may be overwritten by subsequent CREDENTIAL frames. The server must exclusively use the CREDENTIAL vector when evaluating the client certificates associated with an origin. The server may change the size of this vector by sending a SETTINGS frame with the setting SETTINGS_CLIENT_CERTIFICATE_VECTOR_SIZE value specified. In the event that the new size is smaller than the current size, truncation occurs preserving lower-index slots as possible.

TLS renegotiation with client authentication is incompatible with HTTP/2.0 given the multiplexed nature of HTTP/2.0. Specifically, imagine that the client has 2 requests outstanding to the server for two different pages (in different tabs). When the renegotiation + client certificate request comes in, the browser is unable to determine which resource triggered the client certificate request, in order to prompt the user accordingly.



Slot: The index in the server's client certificate vector where this certificate should be stored. If there is already a certificate

stored at this index, it will be overwritten. The index is one based, not zero based; zero is an invalid slot index.

Proof: Cryptographic proof that the client has possession of the private key associated with the certificate. The format is a TLS digitally-signed element ([RFC5246], Section 4.7). The signature algorithm must be the same as that used in the CertificateVerify message. However, since the MD5+SHA1 signature type used in TLS 1.0 connections can not be correctly encoded in a digitally-signed element, SHA1 must be used when MD5+SHA1 was used in the SSL connection. The signature is calculated over a 32 byte TLS extractor value (http://tools.ietf.org/html/rfc5705) with a label of "EXPORTER HTTP/2.0 certificate proof" using the empty string as context. ForRSA certificates the signature would be a PKCS#1 v1.5 signature. For ECDSA, it would be an ECDSA-Sig-Value (http://tools.ietf.org/html/rfc5480#appendix-A). For a 1024-bit RSA key, the CREDENTIAL message would be ~500 bytes.

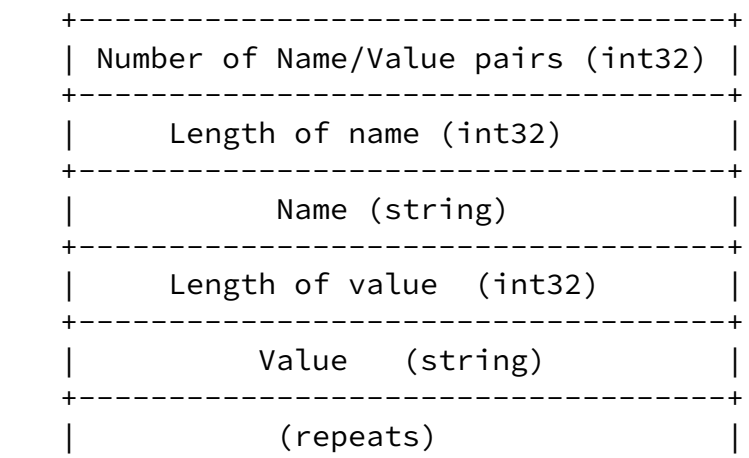
Certificate: The certificate chain, starting with the leaf certificate. Each certificate must be encoded as a 32 bit length, followed by a DER encoded certificate. The certificate must be of the same type (RSA, ECDSA, etc) as the client certificate associated with the SSL connection.

If the server receives a request for a resource with unacceptable credential (either missing or invalid), it must reply with a RST_STREAM frame with the status code INVALID_CREDENTIALS. Upon receipt of a RST_STREAM frame with INVALID_CREDENTIALS, the client should initiate a new stream directly to the requested origin and resend the request. Note, HTTP/2.0 does not allow the server to request different client authentication for different resources in the same origin.

If the server receives an invalid CREDENTIAL frame, it MUST respond with a GOAWAY frame and shutdown the session.

3.6.10. Name/Value Header Block

The Name/Value Header Block is found in the SYN_STREAM, SYN_REPLY and HEADERS control frames, and shares a common format:



Number of Name/Value pairs: The number of repeating name/value pairs following this field.

List of Name/Value pairs:

Length of Name: a 32-bit value containing the number of octets in the name field. Note that in practice, this length must not exceed 2^24, as that is the maximum size of a HTTP/2.0 frame.

Name: 0 or more octets, 8-bit sequences of data, excluding 0.

Length of Value: a 32-bit value containing the number of octets in the value field. Note that in practice, this length must not exceed 2^24, as that is the maximum size of a HTTP/2.0 frame.

Value: 0 or more octets, 8-bit sequences of data, excluding 0.

Each header name must have at least one value. Header names are encoded using the US-ASCII character set [ASCII] and must be all lower case. The length of each name must be greater than zero. A recipient of a zero-length name MUST issue a stream error (Section 3.4.2) with the status code PROTOCOL_ERROR for the stream-id.

Duplicate header names are not allowed. To send two identically named headers, send a header with two values, where the values are separated by a single NUL (0) byte. A header value can either be empty (e.g. the length is zero) or it can contain multiple, NUL-separated values, each with length greater than zero. The value never starts nor ends with a NUL character. Recipients of illegal

value fields MUST issue a stream error (Section 3.4.2) with the status code PROTOCOL_ERROR for the stream-id.

3.6.10.1. Compression

The Name/Value Header Block is a section of the SYN_STREAM, SYN_REPLY, and HEADERS frames used to carry header meta-data. This block is always compressed using zlib compression. Within this specification, any reference to 'zlib' is referring to the ZLIB Compressed Data Format Specification Version 3.3 as part of RFC1950. [RFC1950]

For each HEADERS compression instance, the initial state is initialized using the following dictionary [UDELCOMPRESSION]:

<CODE BEGINS>

```
const unsigned char http2_dictionary_txt[] = {
  0x00, 0x00, 0x00, 0x07, 0x6f, 0x70, 0x74, 0x69,  \\ - - - - o p t i
  0x6f, 0x6e, 0x73, 0x00, 0x00, 0x00, 0x04, 0x68,  \\ o n s - - - - h
  0x65, 0x61, 0x64, 0x00, 0x00, 0x00, 0x04, 0x70,  \\ e a d - - - - p
  0x6f, 0x73, 0x74, 0x00, 0x00, 0x00, 0x03, 0x70,  \\ o s t - - - - p
  0x75, 0x74, 0x00, 0x00, 0x00, 0x06, 0x64, 0x65,  \\ u t - - - - d e
  0x6c, 0x65, 0x74, 0x65, 0x00, 0x00, 0x00, 0x05,  \\ l e t e - - - -
  0x74, 0x72, 0x61, 0x63, 0x65, 0x00, 0x00, 0x00,  \\ t r a c e - - -
  0x06, 0x61, 0x63, 0x63, 0x65, 0x70, 0x74, 0x00,  \\ - a c c e p t -
  0x00, 0x00, 0x0e, 0x61, 0x63, 0x63, 0x65, 0x70,  \\ - - - a c c e p
  0x74, 0x2d, 0x63, 0x68, 0x61, 0x72, 0x73, 0x65,  \\ t - c h a r s e
  0x74, 0x00, 0x00, 0x00, 0x0f, 0x61, 0x63, 0x63,  \\ t - - - - a c c
  0x65, 0x70, 0x74, 0x2d, 0x65, 0x6e, 0x63, 0x6f,  \\ e p t - e n c o
  0x64, 0x69, 0x6e, 0x67, 0x00, 0x00, 0x00, 0x0f,  \\ d i n g - - - -
  0x61, 0x63, 0x63, 0x65, 0x70, 0x74, 0x2d, 0x6c,  \\ a c c e p t - l
  0x61, 0x6e, 0x67, 0x75, 0x61, 0x67, 0x65, 0x00,  \\ a n g u a g e -
  0x00, 0x00, 0x0d, 0x61, 0x63, 0x63, 0x65, 0x70,  \\ - - - a c c e p
  0x74, 0x2d, 0x72, 0x61, 0x6e, 0x67, 0x65, 0x73,  \\ t - r a n g e s
  0x00, 0x00, 0x00, 0x03, 0x61, 0x67, 0x65, 0x00,  \\ - - - - a g e -
  0x00, 0x00, 0x05, 0x61, 0x6c, 0x6c, 0x6f, 0x77,  \\ - - - a l l o w
  0x00, 0x00, 0x00, 0x0d, 0x61, 0x75, 0x74, 0x68,  \\ - - - - a u t h
  0x6f, 0x72, 0x69, 0x7a, 0x61, 0x74, 0x69, 0x6f,  \\ o r i z a t i o
```

```
0x6e, 0x00, 0x00, 0x00, 0x0d, 0x63, 0x61, 0x63,  \\ n - - - - c a c
0x68, 0x65, 0x2d, 0x63, 0x6f, 0x6e, 0x74, 0x72,  \\ h e - c o n t r
0x6f, 0x6c, 0x00, 0x00, 0x00, 0x0a, 0x63, 0x6f,  \\ o l - - - - c o
0x6e, 0x6e, 0x65, 0x63, 0x74, 0x69, 0x6f, 0x6e,  \\ n n e c t i o n
0x00, 0x00, 0x00, 0x0c, 0x63, 0x6f, 0x6e, 0x74,  \\ - - - - c o n t
0x65, 0x6e, 0x74, 0x2d, 0x62, 0x61, 0x73, 0x65,  \\ e n t - b a s e
0x00, 0x00, 0x00, 0x10, 0x63, 0x6f, 0x6e, 0x74,  \\ - - - - c o n t
0x65, 0x6e, 0x74, 0x2d, 0x65, 0x6e, 0x63, 0x6f,  \\ e n t - e n c o
0x64, 0x69, 0x6e, 0x67, 0x00, 0x00, 0x00, 0x10,  \\ d i n g - - - -
0x63, 0x6f, 0x6e, 0x74, 0x65, 0x6e, 0x74, 0x2d,  \\ c o n t e n t -
0x6c, 0x61, 0x6e, 0x67, 0x75, 0x61, 0x67, 0x65,  \\ l a n g u a g e
0x00, 0x00, 0x00, 0x0e, 0x63, 0x6f, 0x6e, 0x74,  \\ - - - - c o n t
```



```
0x65, 0x6e, 0x74, 0x2d, 0x6c, 0x65, 0x6e, 0x67,  \\ e n t - l e n g
0x74, 0x68, 0x00, 0x00, 0x00, 0x10, 0x63, 0x6f,  \\ t h - - - - c o
0x6e, 0x74, 0x65, 0x6e, 0x74, 0x2d, 0x6c, 0x6f,  \\ n t e n t - l o
0x63, 0x61, 0x74, 0x69, 0x6f, 0x6e, 0x00, 0x00,  \\ c a t i o n - -
0x00, 0x0b, 0x63, 0x6f, 0x6e, 0x74, 0x65, 0x6e,  \\ - - c o n t e n
0x74, 0x2d, 0x6d, 0x64, 0x35, 0x00, 0x00, 0x00,  \\ t - m d 5 - - -
0x0d, 0x63, 0x6f, 0x6e, 0x74, 0x65, 0x6e, 0x74,  \\ - c o n t e n t
0x2d, 0x72, 0x61, 0x6e, 0x67, 0x65, 0x00, 0x00,  \\ - r a n g e - -
0x00, 0x0c, 0x63, 0x6f, 0x6e, 0x74, 0x65, 0x6e,  \\ - - c o n t e n
0x74, 0x2d, 0x74, 0x79, 0x70, 0x65, 0x00, 0x00,  \\ t - t y p e - -
0x00, 0x04, 0x64, 0x61, 0x74, 0x65, 0x00, 0x00,  \\ - - d a t e - -
0x00, 0x04, 0x65, 0x74, 0x61, 0x67, 0x00, 0x00,  \\ - - e t a g - -
0x00, 0x06, 0x65, 0x78, 0x70, 0x65, 0x63, 0x74,  \\ - - e x p e c t
0x00, 0x00, 0x00, 0x07, 0x65, 0x78, 0x70, 0x69,  \\ - - - - e x p i
0x72, 0x65, 0x73, 0x00, 0x00, 0x00, 0x04, 0x66,  \\ r e s - - - - f
0x72, 0x6f, 0x6d, 0x00, 0x00, 0x00, 0x04, 0x68,  \\ r o m - - - - h
0x6f, 0x73, 0x74, 0x00, 0x00, 0x00, 0x08, 0x69,  \\ o s t - - - - i
0x66, 0x2d, 0x6d, 0x61, 0x74, 0x63, 0x68, 0x00,  \\ f - m a t c h -
0x00, 0x00, 0x11, 0x69, 0x66, 0x2d, 0x6d, 0x6f,  \\ - - - i f - m o
0x64, 0x69, 0x66, 0x69, 0x65, 0x64, 0x2d, 0x73,  \\ d i f i e d - s
0x69, 0x6e, 0x63, 0x65, 0x00, 0x00, 0x00, 0x0d,  \\ i n c e - - - -
0x69, 0x66, 0x2d, 0x6e, 0x6f, 0x6e, 0x65, 0x2d,  \\ i f - n o n e -
0x6d, 0x61, 0x74, 0x63, 0x68, 0x00, 0x00, 0x00,  \\ m a t c h - - -
0x08, 0x69, 0x66, 0x2d, 0x72, 0x61, 0x6e, 0x67,  \\ - i f - r a n g
0x65, 0x00, 0x00, 0x00, 0x13, 0x69, 0x66, 0x2d,  \\ e - - - - i f -
0x75, 0x6e, 0x6d, 0x6f, 0x64, 0x69, 0x66, 0x69,  \\ u n m o d i f i
0x65, 0x64, 0x2d, 0x73, 0x69, 0x6e, 0x63, 0x65,  \\ e d - s i n c e
0x00, 0x00, 0x00, 0x0d, 0x6c, 0x61, 0x73, 0x74,  \\ - - - - l a s t
0x2d, 0x6d, 0x6f, 0x64, 0x69, 0x66, 0x69, 0x65,  \\ - m o d i f i e
```

```
0x64, 0x00, 0x00, 0x00, 0x08, 0x6c, 0x6f, 0x63,  \\ d - - - - l o c
0x61, 0x74, 0x69, 0x6f, 0x6e, 0x00, 0x00, 0x00,  \\ a t i o n - - -
0x0c, 0x6d, 0x61, 0x78, 0x2d, 0x66, 0x6f, 0x72,  \\ - m a x - f o r
0x77, 0x61, 0x72, 0x64, 0x73, 0x00, 0x00, 0x00,  \\ w a r d s - - -
0x06, 0x70, 0x72, 0x61, 0x67, 0x6d, 0x61, 0x00,  \\ - p r a g m a -
0x00, 0x00, 0x12, 0x70, 0x72, 0x6f, 0x78, 0x79,  \\ - - - p r o x y
0x2d, 0x61, 0x75, 0x74, 0x68, 0x65, 0x6e, 0x74,  \\ - a u t h e n t
0x69, 0x63, 0x61, 0x74, 0x65, 0x00, 0x00, 0x00,  \\ i c a t e - - -
0x13, 0x70, 0x72, 0x6f, 0x78, 0x79, 0x2d, 0x61,  \\ - p r o x y - a
0x75, 0x74, 0x68, 0x6f, 0x72, 0x69, 0x7a, 0x61,  \\ u t h o r i z a
0x74, 0x69, 0x6f, 0x6e, 0x00, 0x00, 0x00, 0x05,  \\ t i o n - - - -
0x72, 0x61, 0x6e, 0x67, 0x65, 0x00, 0x00, 0x00,  \\ r a n g e - - -
0x07, 0x72, 0x65, 0x66, 0x65, 0x72, 0x65, 0x72,  \\ - r e f e r e r
0x00, 0x00, 0x00, 0x0b, 0x72, 0x65, 0x74, 0x72,  \\ - - - - r e t r
0x79, 0x2d, 0x61, 0x66, 0x74, 0x65, 0x72, 0x00,  \\ y - a f t e r -
0x00, 0x00, 0x06, 0x73, 0x65, 0x72, 0x76, 0x65,  \\ - - - s e r v e
0x72, 0x00, 0x00, 0x00, 0x02, 0x74, 0x65, 0x00,  \\ r - - - - t e -
0x00, 0x00, 0x07, 0x74, 0x72, 0x61, 0x69, 0x6c,  \\ - - - t r a i l
0x65, 0x72, 0x00, 0x00, 0x00, 0x11, 0x74, 0x72,  \\ e r - - - - t r
```



```
0x61, 0x6e, 0x73, 0x66, 0x65, 0x72, 0x2d, 0x65,  \\ a n s f e r - e
0x6e, 0x63, 0x6f, 0x64, 0x69, 0x6e, 0x67, 0x00,  \\ n c o d i n g -
0x00, 0x00, 0x07, 0x75, 0x70, 0x67, 0x72, 0x61,  \\ - - - u p g r a
0x64, 0x65, 0x00, 0x00, 0x00, 0x0a, 0x75, 0x73,  \\ d e - - - - u s
0x65, 0x72, 0x2d, 0x61, 0x67, 0x65, 0x6e, 0x74,  \\ e r - a g e n t
0x00, 0x00, 0x00, 0x04, 0x76, 0x61, 0x72, 0x79,  \\ - - - - v a r y
0x00, 0x00, 0x00, 0x03, 0x76, 0x69, 0x61, 0x00,  \\ - - - - v i a -
0x00, 0x00, 0x07, 0x77, 0x61, 0x72, 0x6e, 0x69,  \\ - - - w a r n i
0x6e, 0x67, 0x00, 0x00, 0x00, 0x10, 0x77, 0x77,  \\ n g - - - - w w
0x77, 0x2d, 0x61, 0x75, 0x74, 0x68, 0x65, 0x6e,  \\ w - a u t h e n
0x74, 0x69, 0x63, 0x61, 0x74, 0x65, 0x00, 0x00,  \\ t i c a t e - -
0x00, 0x06, 0x6d, 0x65, 0x74, 0x68, 0x6f, 0x64,  \\ - - m e t h o d
0x00, 0x00, 0x00, 0x03, 0x67, 0x65, 0x74, 0x00,  \\ - - - - g e t -
0x00, 0x00, 0x06, 0x73, 0x74, 0x61, 0x74, 0x75,  \\ - - - s t a t u
0x73, 0x00, 0x00, 0x00, 0x06, 0x32, 0x30, 0x30,  \\ s - - - - 2 0 0
0x20, 0x4f, 0x4b, 0x00, 0x00, 0x00, 0x07, 0x76,  \\ - O K - - - - v
0x65, 0x72, 0x73, 0x69, 0x6f, 0x6e, 0x00, 0x00,  \\ e r s i o n - -
0x00, 0x08, 0x48, 0x54, 0x54, 0x50, 0x2f, 0x31,  \\ - - H T T P - 1
0x2e, 0x31, 0x00, 0x00, 0x00, 0x03, 0x75, 0x72,  \\ - 1 - - - - u r
0x6c, 0x00, 0x00, 0x00, 0x06, 0x70, 0x75, 0x62,  \\ l - - - - p u b
0x6c, 0x69, 0x63, 0x00, 0x00, 0x00, 0x0a, 0x73,  \\ l i c - - - - s
0x65, 0x74, 0x2d, 0x63, 0x6f, 0x6f, 0x6b, 0x69,  \\ e t - c o o k i
```

```
0x65, 0x00, 0x00, 0x00, 0x0a, 0x6b, 0x65, 0x65,  \\ e - - - - k e e
0x70, 0x2d, 0x61, 0x6c, 0x69, 0x76, 0x65, 0x00,  \\ p - a l i v e -
0x00, 0x00, 0x06, 0x6f, 0x72, 0x69, 0x67, 0x69,  \\ - - - o r i g i
0x6e, 0x31, 0x30, 0x30, 0x31, 0x30, 0x31, 0x32,  \\ n 1 0 0 1 0 1 2
0x30, 0x31, 0x32, 0x30, 0x32, 0x32, 0x30, 0x35,  \\ 0 1 2 0 2 2 0 5
0x32, 0x30, 0x36, 0x33, 0x30, 0x30, 0x33, 0x30,  \\ 2 0 6 3 0 0 3 0
0x32, 0x33, 0x30, 0x33, 0x33, 0x30, 0x34, 0x33,  \\ 2 3 0 3 3 0 4 3
0x30, 0x35, 0x33, 0x30, 0x36, 0x33, 0x30, 0x37,  \\ 0 5 3 0 6 3 0 7
0x34, 0x30, 0x32, 0x34, 0x30, 0x35, 0x34, 0x30,  \\ 4 0 2 4 0 5 4 0
0x36, 0x34, 0x30, 0x37, 0x34, 0x30, 0x38, 0x34,  \\ 6 4 0 7 4 0 8 4
0x30, 0x39, 0x34, 0x31, 0x30, 0x34, 0x31, 0x31,  \\ 0 9 4 1 0 4 1 1
0x34, 0x31, 0x32, 0x34, 0x31, 0x33, 0x34, 0x31,  \\ 4 1 2 4 1 3 4 1
0x34, 0x34, 0x31, 0x35, 0x34, 0x31, 0x36, 0x34,  \\ 4 4 1 5 4 1 6 4
0x31, 0x37, 0x35, 0x30, 0x32, 0x35, 0x30, 0x34,  \\ 1 7 5 0 2 5 0 4
0x35, 0x30, 0x35, 0x32, 0x30, 0x33, 0x20, 0x4e,  \\ 5 0 5 2 0 3 - N
0x6f, 0x6e, 0x2d, 0x41, 0x75, 0x74, 0x68, 0x6f,  \\ o n - A u t h o
0x72, 0x69, 0x74, 0x61, 0x74, 0x69, 0x76, 0x65,  \\ r i t a t i v e
0x20, 0x49, 0x6e, 0x66, 0x6f, 0x72, 0x6d, 0x61,  \\ - I n f o r m a
0x74, 0x69, 0x6f, 0x6e, 0x32, 0x30, 0x34, 0x20,  \\ t i o n 2 0 4 -
0x4e, 0x6f, 0x20, 0x43, 0x6f, 0x6e, 0x74, 0x65,  \\ N o - C o n t e
0x6e, 0x74, 0x33, 0x30, 0x31, 0x20, 0x4d, 0x6f,  \\ n t 3 0 1 - M o
0x76, 0x65, 0x64, 0x20, 0x50, 0x65, 0x72, 0x6d,  \\ v e d - P e r m
0x61, 0x6e, 0x65, 0x6e, 0x74, 0x6c, 0x79, 0x34,  \\ a n e n t l y 4
0x30, 0x30, 0x20, 0x42, 0x61, 0x64, 0x20, 0x52,  \\ 0 0 - B a d - R
0x65, 0x71, 0x75, 0x65, 0x73, 0x74, 0x34, 0x30,  \\ e q u e s t 4 0
0x31, 0x20, 0x55, 0x6e, 0x61, 0x75, 0x74, 0x68,  \\ 1 - U n a u t h
```



```
0x6f, 0x72, 0x69, 0x7a, 0x65, 0x64, 0x34, 0x30,  \\ o r i z e d 4 0
0x33, 0x20, 0x46, 0x6f, 0x72, 0x62, 0x69, 0x64,  \\ 3 - F o r b i d
0x64, 0x65, 0x6e, 0x34, 0x30, 0x34, 0x20, 0x4e,  \\ d e n 4 0 4 - N
0x6f, 0x74, 0x20, 0x46, 0x6f, 0x75, 0x6e, 0x64,  \\ o t - F o u n d
0x35, 0x30, 0x30, 0x20, 0x49, 0x6e, 0x74, 0x65,  \\ 5 0 0 - I n t e
0x72, 0x6e, 0x61, 0x6c, 0x20, 0x53, 0x65, 0x72,  \\ r n a l - S e r
0x76, 0x65, 0x72, 0x20, 0x45, 0x72, 0x72, 0x6f,  \\ v e r - E r r o
0x72, 0x35, 0x30, 0x31, 0x20, 0x4e, 0x6f, 0x74,  \\ r 5 0 1 - N o t
0x20, 0x49, 0x6d, 0x70, 0x6c, 0x65, 0x6d, 0x65,  \\ - I m p l e m e
0x6e, 0x74, 0x65, 0x64, 0x35, 0x30, 0x33, 0x20,  \\ n t e d 5 0 3 -
0x53, 0x65, 0x72, 0x76, 0x69, 0x63, 0x65, 0x20,  \\ S e r v i c e -
0x55, 0x6e, 0x61, 0x76, 0x61, 0x69, 0x6c, 0x61,  \\ U n a v a i l a
0x62, 0x6c, 0x65, 0x4a, 0x61, 0x6e, 0x20, 0x46,  \\ b l e J a n - F
0x65, 0x62, 0x20, 0x4d, 0x61, 0x72, 0x20, 0x41,  \\ e b - M a r - A
0x70, 0x72, 0x20, 0x4d, 0x61, 0x79, 0x20, 0x4a,  \\ p r - M a y - J
```

```
  0x75, 0x6e, 0x20, 0x4a, 0x75, 0x6c, 0x20, 0x41,  \\ u n - J u l - A
  0x75, 0x67, 0x20, 0x53, 0x65, 0x70, 0x74, 0x20,  \\ u g - S e p t -
  0x4f, 0x63, 0x74, 0x20, 0x4e, 0x6f, 0x76, 0x20,  \\ O c t - N o v -
  0x44, 0x65, 0x63, 0x20, 0x30, 0x30, 0x3a, 0x30,  \\ D e c - 0 0 - 0
  0x30, 0x3a, 0x30, 0x30, 0x20, 0x4d, 0x6f, 0x6e,  \\ 0 - 0 0 - M o n
  0x2c, 0x20, 0x54, 0x75, 0x65, 0x2c, 0x20, 0x57,  \\ - - T u e - - W
  0x65, 0x64, 0x2c, 0x20, 0x54, 0x68, 0x75, 0x2c,  \\ e d - - T h u -
  0x20, 0x46, 0x72, 0x69, 0x2c, 0x20, 0x53, 0x61,  \\ - F r i - - S a
  0x74, 0x2c, 0x20, 0x53, 0x75, 0x6e, 0x2c, 0x20,  \\ t - - S u n - -
  0x47, 0x4d, 0x54, 0x63, 0x68, 0x75, 0x6e, 0x6b,  \\ G M T c h u n k
  0x65, 0x64, 0x2c, 0x74, 0x65, 0x78, 0x74, 0x2f,  \\ e d - t e x t -
  0x68, 0x74, 0x6d, 0x6c, 0x2c, 0x69, 0x6d, 0x61,  \\ h t m l - i m a
  0x67, 0x65, 0x2f, 0x70, 0x6e, 0x67, 0x2c, 0x69,  \\ g e - p n g - i
  0x6d, 0x61, 0x67, 0x65, 0x2f, 0x6a, 0x70, 0x67,  \\ m a g e - j p g
  0x2c, 0x69, 0x6d, 0x61, 0x67, 0x65, 0x2f, 0x67,  \\ - i m a g e - g
  0x69, 0x66, 0x2c, 0x61, 0x70, 0x70, 0x6c, 0x69,  \\ i f - a p p l i
  0x63, 0x61, 0x74, 0x69, 0x6f, 0x6e, 0x2f, 0x78,  \\ c a t i o n - x
  0x6d, 0x6c, 0x2c, 0x61, 0x70, 0x70, 0x6c, 0x69,  \\ m l - a p p l i
  0x63, 0x61, 0x74, 0x69, 0x6f, 0x6e, 0x2f, 0x78,  \\ c a t i o n - x
  0x68, 0x74, 0x6d, 0x6c, 0x2b, 0x78, 0x6d, 0x6c,  \\ h t m l - x m l
  0x2c, 0x74, 0x65, 0x78, 0x74, 0x2f, 0x70, 0x6c,  \\ - t e x t - p l
  0x61, 0x69, 0x6e, 0x2c, 0x74, 0x65, 0x78, 0x74,  \\ a i n - t e x t
  0x2f, 0x6a, 0x61, 0x76, 0x61, 0x73, 0x63, 0x72,  \\ - j a v a s c r
  0x69, 0x70, 0x74, 0x2c, 0x70, 0x75, 0x62, 0x6c,  \\ i p t - p u b l
  0x69, 0x63, 0x70, 0x72, 0x69, 0x76, 0x61, 0x74,  \\ i c p r i v a t
  0x65, 0x6d, 0x61, 0x78, 0x2d, 0x61, 0x67, 0x65,  \\ e m a x - a g e
  0x3d, 0x67, 0x7a, 0x69, 0x70, 0x2c, 0x64, 0x65,  \\ - g z i p - d e
  0x66, 0x6c, 0x61, 0x74, 0x65, 0x2c, 0x73, 0x64,  \\ f l a t e - s d
  0x63, 0x68, 0x63, 0x68, 0x61, 0x72, 0x73, 0x65,  \\ c h c h a r s e
  0x74, 0x3d, 0x75, 0x74, 0x66, 0x2d, 0x38, 0x63,  \\ t - u t f - 8 c
  0x68, 0x61, 0x72, 0x73, 0x65, 0x74, 0x3d, 0x69,  \\ h a r s e t - i
  0x73, 0x6f, 0x2d, 0x38, 0x38, 0x35, 0x39, 0x2d,  \\ s o - 8 8 5 9 -
  0x31, 0x2c, 0x75, 0x74, 0x66, 0x2d, 0x2c, 0x2a,  \\ 1 - u t f - - -
```



```
  0x2c, 0x65, 0x6e, 0x71, 0x3d, 0x30, 0x2e         \\ - e n q - 0 -
};
```

<CODE ENDS>

The entire contents of the name/value header block is compressed using zlib. There is a single zlib stream for all name value pairs in one direction on a connection. HTTP/2.0 uses a SYNC_FLUSH between

each compressed frame.

Implementation notes: the compression engine can be tuned to favor speed or size. Optimizing for size increases memory use and CPU consumption. Because header blocks are generally small, implementors may want to reduce the window-size of the compression engine from the default 15bits (a 32KB window) to more like 11bits (a 2KB window). The exact setting is chosen by the compressor, the decompressor will work with any setting.

4. HTTP Layering over HTTP/2.0

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 the HTTP/1.1 specification in RFC2616 [RFC2616] 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 (a new SYN_STREAM) 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 determinstically 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 any SYN_STREAMs from the remote, the GOAWAY will contain a last-stream-id of 0.

4.2. HTTP Request/Response

4.2.1. Request

The client initiates a request by sending a SYN_STREAM frame. For requests which do not contain a body, the SYN_STREAM frame MUST set the FLAG_FIN, indicating that the client intends to send no further data on this stream. For requests which do contain a body, the SYN_STREAM will not contain the FLAG_FIN, and the body will follow the SYN_STREAM in a series of DATA frames. The last DATA frame will set the FLAG_FIN to indicate the end of the body.

The SYN_STREAM Name/Value section will contain all of the HTTP headers which 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 first line of the request is unfolded into name/value pairs like other HTTP headers and MUST be present:

":method" - the HTTP method for this request (e.g. "GET", "POST", "HEAD", etc)

":path" - the url-path for this url with "/" prefixed. (See RFC1738 [RFC1738]). For example, for "http://www.google.com/search?q=dogs" the path would be "/search?q=dogs".

":version" - the HTTP version of this request (e.g. "HTTP/1.1")

In addition, the following two name/value pairs must also be present in every request:

":host" - the hostport (See RFC1738 [RFC1738]) portion of the URL for this request (e.g. "www.google.com:1234"). This header is the same as the HTTP 'Host' header.

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

Header names are all lowercase.

The Connection, Host, Keep-Alive, Proxy-Connection, and Transfer-Encoding headers 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.

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, the server MUST return a 400 (Bad Request) error.

POST-specific changes:

Although POSTs are inherently chunked, POST requests SHOULD also be accompanied by a Content-Length header. There are two reasons for this: First, it assists with upload progress meters for an improved user experience. But second, we know from early versions of HTTP/2.0 that failure to send a content length header is incompatible with many existing HTTP server implementations. Existing user-agents do not omit the Content-Length header, and server implementations have come to depend upon this.

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 SYN_STREAM without all of the method, host, path, scheme, and version headers, the server MUST reply with a HTTP 400 Bad Request reply.

4.2.2. Response

The server responds to a client request with a SYN_REPLY frame. Symmetric to the client's upload stream, server will send data after the SYN_REPLY frame via a series of DATA frames, and the last data frame will contain the FLAG_FIN to indicate successful end-of-stream. If a response (like a 202 or 204 response) contains no body, the SYN_REPLY frame may contain the FLAG_FIN 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 headers and must be present:

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

":version" - The HTTP response version (e.g. "HTTP/1.1")

All header names must be lowercase.

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

Responses MAY be accompanied by a Content-Length header for advisory purposes. (e.g. for UI progress meters)

If a client receives a response where the sum of the data frame payload lengths does not equal the size of the Content-Length header, the client MUST ignore the content length header.

If a client receives a SYN_REPLY without a status or without a version header, the client must reply with a RST_STREAM frame indicating a PROTOCOL ERROR.

4.2.3. Authentication

When a client sends a request to an origin server that requires authentication, the server can reply with a "401 Unauthorized" response, and include a WWW-Authenticate challenge header that

defines the authentication scheme to be used. The client then retries the request with an Authorization header appropriate to the specified authentication scheme.

There are four options for proxy authentication, Basic, Digest, NTLM and Negotiate (SPNEGO). The first two options were defined in RFC2617 [RFC2617], and are stateless. The second two options were developed by Microsoft and specified in RFC4559 [RFC4559], and are stateful; otherwise known as multi-round authentication, or

connection authentication.

4.2.3.1. Stateless Authentication

Stateless Authentication over HTTP/2.0 is identical to how it is performed over HTTP. If multiple HTTP/2.0 streams are concurrently sent to a single server, each will authenticate independently, similar to how two HTTP connections would independently authenticate to a proxy server.

4.2.3.2. Stateful Authentication

Unfortunately, the stateful authentication mechanisms were implemented and defined in a such a way that directly violates RFC2617 - they do not include a "realm" as part of the request. This is problematic in HTTP/2.0 because it makes it impossible for a client to disambiguate two concurrent server authentication challenges.

To deal with this case, HTTP/2.0 servers using Stateful Authentication MUST implement one of two changes:

Servers can add a "realm=<desired realm>" header so that the two authentication requests can be disambiguated and run concurrently. Unfortunately, given how these mechanisms work, this is probably not practical.

Upon sending the first stateful challenge response, the server MUST buffer and defer all further frames which are not part of completing the challenge until the challenge has completed. Completing the authentication challenge may take multiple round trips. Once the client receives a "401 Authenticate" response for

a stateful authentication type, it MUST stop sending new requests to the server until the authentication has completed by receiving a non-401 response on at least one stream.

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.

Browsers receiving a pushed response MUST validate that the server is authorized to push the URL using the browser same-origin [RFC6454] policy. For example, a HTTP/2.0 connection to www.foo.com is generally not permitted to push a response for www.evil.com.

If the browser accepts a pushed response (e.g. it does not send a RST_STREAM), the browser MUST attempt to cache the pushed response in same way that it would cache any other response. This means validating the response headers and inserting into the disk cache.

Because pushed responses have no request, they have no request headers associated with them. At the framing layer, HTTP/2.0 pushed streams contain an "associated-stream-id" which indicates the requested stream for which the pushed stream is related. The pushed stream inherits all of the headers from the associated-stream-id with the exception of ":host", ":scheme", and ":path", which are provided as part of the pushed response stream headers. The browser MUST store these inherited and implied request headers 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.

4.3.1. Server implementation

When the server intends to push a resource to the user-agent, it opens a new stream by sending a unidirectional SYN_STREAM. The SYN_STREAM MUST include an Associated-To-Stream-ID, and MUST set the FLAG_UNIDIRECTIONAL flag. The SYN_STREAM MUST include headers for ":scheme", ":host", ":path", which represent the URL for the resource being pushed. Subsequent headers may follow in HEADERS frames. The purpose of the association is so that the user-agent can differentiate which request induced the pushed stream; without it, if the user-agent had two tabs open to the same page, each pushing unique content under a fixed URL, the user-agent would not be able to differentiate the requests.

The Associated-To-Stream-ID must be the ID of an existing, open stream. The reason for this restriction is to have a clear endpoint for pushed content. If the user-agent requested a resource on stream 11, the server replies on stream 11. It can push any number of additional streams to the client before sending a FLAG_FIN on stream 11. However, once the originating stream is closed no further push streams may be associated with it. The pushed streams do not need to be closed (FIN set) before the originating stream is closed, they only need to be created before the originating stream closes.

It is illegal for a server to push a resource with the Associated-To-Stream-ID of 0.

To minimize race conditions with the client, the SYN_STREAM for the pushed resources MUST be sent prior to sending any content which could allow the client to discover the pushed resource and request it.

The server MUST only push resources which would have been returned from a GET request.

Note: If the server does not have all of the Name/Value Response headers available at the time it issues the HEADERS frame for the pushed resource, it may later use an additional HEADERS frame to augment the name/value pairs to be associated with the pushed stream. The subsequent HEADERS frame(s) must not contain a header for ':host', ':scheme', or ':path' (e.g. the server can't change the identity of the resource to be pushed). The HEADERS frame must not

contain duplicate headers with a previously sent HEADERS frame. The server must send a HEADERS frame including the scheme/host/port headers before sending any data frames on the stream.

4.3.2. Client implementation

When fetching a resource the client has 3 possibilities:

the resource is not being pushed

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

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

When a SYN_STREAM and HEADERS frame which contains an Associated-To-Stream-ID is received, the client must not issue GET requests for the resource in the pushed stream, and instead wait for the pushed stream to arrive.

If a client receives a server push stream with stream-id 0, it MUST issue a session error (Section 3.4.1) with the status code PROTOCOL_ERROR.

When a client receives a SYN_STREAM from the server without a the ':host', ':scheme', and ':path' headers in the Name/Value section, it MUST reply with a RST_STREAM with error code HTTP_PROTOCOL_ERROR.

To cancel individual server push streams, the client can issue a stream error (Section 3.4.2) with error code CANCEL. Upon receipt, the server MUST stop sending on this stream immediately (this is an

Abrupt termination).

To cancel all server push streams related to a request, the client may issue a stream error (Section 3.4.2) with error code 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.

If the server sends a HEADER frame containing duplicate headers with a previous HEADERS frame for the same stream, the client must issue a stream error (Section 3.4.2) with error code 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 headers (http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.40).

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, the definition of the HEADERS and compression contexts 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 occuring 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 choses 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. Compression Context(s)

When isolating the compression contexts used for communicating with multiple origins, we had a few choices to make. We could have maintained a map (or list) of compression contexts usable for each origin. The basic case is easy - each HEADERS frame would need to identify the context to use for that frame. However, compression contexts are not cheap, so the lifecycle of each context would need to be bounded. For proxy servers, where we could churn through many contexts, this would be a concern. We considered using a static set of contexts, say 16 of them, which would bound the memory use. We also considered dynamic contexts, which could be created on the fly, and would need to be subsequently destroyed. All of these are complicated, and ultimately we decided that such a mechanism creates too many problems to solve.

Alternatively, we've chosen the simple approach, which is to simply provide a flag for resetting the compression context. For the common case (no proxy), this fine because most requests are to the same origin and we never need to reset the context. For cases where we are using two different origins over a single HTTP/2.0 session, we simply reset the compression state between each transition.

5.6. Unidirectional streams

Many readers notice that unidirectional streams are both a bit confusing in concept and also somewhat redundant. If the recipient of a stream doesn't wish to send data on a stream, it could simply send a SYN_REPLY with the FLAG_FIN bit set. The FLAG_UNIDIRECTIONAL is, therefore, not necessary.

It is true that we don't need the UNIDIRECTIONAL markings. It is added because it avoids the recipient of pushed streams from needing to send a set of empty frames (e.g. the SYN_STREAM w/ FLAG_FIN) which otherwise serve no purpose.

5.7. Data Compression

Generic compression of data portion of the streams (as opposed to compression of the headers) without knowing the content of the stream is redundant. There is no value in compressing a stream which is already compressed. Because of this, HTTP/2.0 does allow data compression to be optional. We included it because study of existing websites shows that many sites are not using compression as they should, and users suffer because of it. We wanted a mechanism where,

at the HTTP/2.0 layer, site administrators could simply force

compression - it is better to compress twice than to not compress.

Overall, however, with this feature being optional and sometimes redundant, it is unclear if it is useful at all. We will likely remove it from the specification.

5.8. 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] in all cases where verification of content is required.

6.2. HTTP Headers and HTTP/2.0 Headers

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

6.3. 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.

6.4. Server Push Implicit Headers

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

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, such as Google Chrome's "incognito mode", may wish to disable client-persisted SETTINGS storage.

Clients MUST clear persisted SETTINGS information when clearing the cookies.

TODO: Put range maximums on each type of setting to limit inappropriate uses.

8. Requirements Notation

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].

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 principles)

- Mark Nottingham and Julian Reschke

10. Normative References

[ASCII] "US-ASCII. Coded Character Set - 7-Bit American Standard Code for Information Interchange. Standard ANSI X3.4-1986, ANSI, 1986.".

[HTTP-p1] Fielding, R. and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", draft-ietf-httpbis-p1-messaging-21 (work in progress), October 2012.

[HTTP-p2] Fielding, R. and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", draft-ietf-httpbis-p2-semantics-21 (work in progress), October 2012.

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

[RFC1738] Berners-Lee, T., Masinter, L., and M. McCahill, "Uniform Resource Locators (URL)", RFC 1738, December 1994.

[RFC1950] Deutsch, L. and J. Gailly, "ZLIB Compressed Data Format Specification version 3.3", RFC 1950, May 1996.

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

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[RFC2617] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and L. Stewart, "HTTP Authentication: Basic and Digest Access Authentication", RFC 2617, June 1999.

[RFC4559] Jaganathan, K., Zhu, L., and J. Brezak, "SPNEGO-based Kerberos and NTLM HTTP Authentication in Microsoft Windows", RFC 4559, June 2006.

[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., "TLS Next Protocol Negotiation", draft-agl-tls-nextprotoneg-01 (work in progress), August 2010.

[UDELCOMPRESSION] Yang, F., Amer, P., and J. Leighton, "A Methodology to Derive SPDY's Initial Dictionary for Zlib Compression", <http://www.eecis.udel.edu/~amer/PEL/poc/pdf/SPDY-Fan.pdf>.

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

A.1. 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 3.5.1) based on <http://tools.ietf.org/html/draft-montenegro-httpbis-http2-fc-principles-01>.

A.2. 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