

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: July 10, 2014

M. Piatek
W. Chan
Google
January 6, 2014

HTTP/2 Stream Dependencies
draft-chan-http2-stream-dependencies-00

Abstract

The existing HTTP/2 prioritization scheme relies purely on integer values to indicate priorities. This simple scheme misses critical support for priority grouping, and does not support other features like resource ordering. This draft proposes using stream dependencies to solve the lack of priority grouping, as well as provide other features.

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 10, 2014.

Copyright Notice

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

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

described in the Simplified BSD License.

Table of Contents

1. Introduction	3
2. Motivation	3
3. Protocol Changes	4
3.1. HEADERS frame	4
3.2. PRIORITY frame	5
3.3. END_STREAM_ACK frame	6
4. Protocol invariants and definitions	6
5. Examples	8
5.1. Specifying an ordering of resource transfers and reacting to document parsing	10
5.2. Servicing multiple tabs/users over a single HTTP/2 connection	13
5.3. Server Push	13
6. Policy Considerations	13
6.1. Assigning and updating dependencies	13
6.2. Server scheduling	14
6.3. Garbage collecting dependency information	14
7. Security Considerations	15
8. Informative References	15
Appendix A. Acknowledgements	15
Authors' Addresses	15

1. Introduction

This document proposes changes to HTTP/2 to support stream dependencies. During a pageload, the server uses dependencies to improve performance by allocating bandwidth capacity to the most important resource transfers first.

The remainder of this document describes the motivation for dependencies, protocol changes to support them, and examples of how those mechanisms can be used by the browser. We conclude with a discussion of the client and server policies afforded by expressing dependency information in HTTP/2.

(Note that flow control is the subject of a separate document and is out of scope here.)

2. Motivation

Dependencies allow an HTTP/2 server to allocate bandwidth capacity efficiently in several common use-cases:

Specifying an ordering of resource transfers

Sharing bandwidth between resources transfer often degrades performance, e.g., when transferring two Javascript resources that cannot be executed until transfer is complete, or two video chunks that will be played back-to-back. In these circumstances, the browser may wish to specify an ordering --- HTML before script1.js before script2.js, for example, or video_chunk1 before video_chunk2.

Reacting to document parsing

Because the browser's document parser blocks while waiting for script and style resource transfers to complete, many resource requests will be issued by simply scanning the tokenized HTML. (For more background, see [PRELOADSCANNER])

As the document parser proceeds, it may learn of higher priority resources. For example, if a script a.js uses document.write to embed another script, b.js, the transfer of b.js should preempt other in-flight resource transfers since the receipt of b.js blocks page layout. Similarly, image transfers that will be styled with display: none should be deferred to prioritize visible content.

Reacting to user behavior

In the case of HTTP/2 proxies, a single TCP connection may multiplex several sites in several tabs. Changing tabs may reorder the relative importance of outstanding streams, e.g., concurrent AJAX requests or page loads. Similarly, a proxy server may coalesce streams to a common origin onto a single connection. As the set of outstanding requests and users changes, the relative importance of each user's streams may change as well.

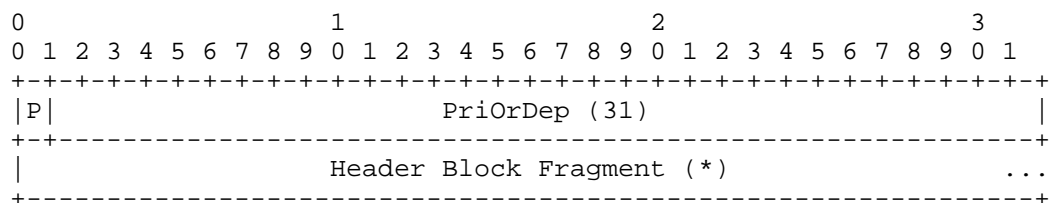
Server Push

Server push can improve performance by eliminating round trips, but it may degrade performance if a pushed stream preempts a more important transfer. For example, a Javascript transfer may block layout and be high priority, or it may be a low-priority async request. Dependencies provide a hint to the server about the relative importance of pushed resources.

3. Protocol Changes

Dependencies are expressed using the existing optional priority field the HEADERS frames and in PRIORITY frames. To ensure clients and servers have consistent view of active streams, we propose the FIN_ACK frame. The section concludes with a set of invariants that clients and servers must maintain when using these frames.

3.1. HEADERS frame



HEADERS Frame Payload

The HEADERS frame defines the following flags:

ORDERED (0x10): Bit 5 being set indicates that the dependency specified by PriOrDep is ordered. If this flag is unset, any dependency is treated as unordered.

Here, the 4 octets previously used by the unused bit and 31 bit Priority field in the HEADERS frame are reinterpreted. The unused bit is now known as the P bit, and the 31 bit Priority field is now

PriOrDep.

P: A bit indicating whether the following PriOrDep bits specify a priority (P = 1) or a stream ID (P = 0) on which this new stream depends.

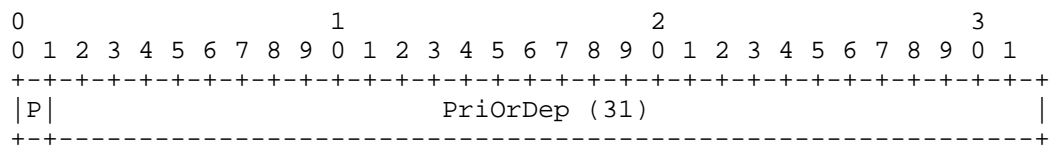
PriOrDep: Depending on the value of P, either the priority of the new stream or a stream ID on which this new stream depends.

The structure and semantics of the Header Block Fragment are unchanged.

P is exclusive; a stream may be assigned a priority or a parent dependency upon creation, but not both. If P = 0 and PriOrDep indicates a dependency; the value MUST correspond to an active stream.

Server push streams are assigned a priority or dependency id at the discretion of the server.

3.2. PRIORITY frame



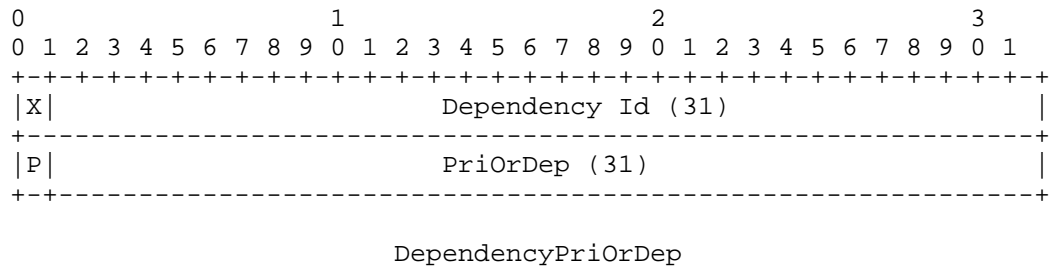
PRIORITY Frame Payload

The `PRIORITY` frame defines the following flags:

ORDERED (0x10): Bit 5 being set indicates that the dependency specified by PriOrDep is ordered. If this flag is unset, any dependency is treated as unordered.

As in HEADERS, the Priority field is changed to be a P/PriOrDep field indicating an update to the 31 bit Dependency Id specified in the header. We relabel the typical Stream Id here as Dependency Id to distinguish it as a referent.

To support batched updates of dependencies, an optional list of `DependencyPriOrDep` pairs with identical semantics may follow. The number of such pairs is determined by examining the frame length.



3.3. END_STREAM_ACK frame

The `END_STREAM_ACK` frame has no payload. It is sent by a client to a server after receiving a frame with the `END_STREAM` flag set. The frame is used to ensure a consistent set of active streams between the client and the server. Consistency is required to maintain the protocol invariants described below.

4. Protocol invariants and definitions

Each stream has at most one dependency. An update to a stream's dependent stream id replaces any existing dependency for the claimant. Specifying multiple dependency ids for a single stream in a PRIORITY frame is a protocol error.

Each stream is depended on by at most one stream. An update to a stream's dependent stream id replaces any existing dependency on the target. Repeating a single dependency id in a PRIORITY frame is a protocol error.

Each dependency has a type: ordered or unordered. Ordered dependencies indicate a sequential transfer preference with respect to the dependent stream id. Unordered dependencies indicate a concurrent transfer preference for the range of the dependency list with unordered dependency links.

For example, where `<-` indicates an ordered dependency and `-` indicates an unordered dependency

```
a.htm <- a.js <- 1.png - 2.png
```

indicates that a.html should preempt a.js which itself should preempt 1.png and 2.png, each of which should transfer concurrently, sharing capacity.

All frames with the `END_STREAM` flag set **MUST** be explicitly acknowledged by clients. To ensure that the client and server have an identical view of active stream ids when specifying dependencies, we require that clients explicitly acknowledge frames with the `END_STREAM` flag set by sending `END_STREAM_ACK`. Servers **MUST** retain dependency relationships for a stream until its `END_STREAM_ACK` is received (or the session is closed). Explicit acknowledgements obviate timeouts for garbage collecting dependency state and enable clients and servers to have a consistent view of dependency relationships.

A dependency id **MUST** correspond to an active stream id. An active stream id is one for which the client has not yet sent an `END_STREAM_ACK` frame. It is a protocol error to name a stream id as a dependency that is not active.

If a server receives an `END_STREAM_ACK` for a stream X on which another stream Y depends, it **SHOULD** update the dependency pointer for Y to reflect the removal of X. The rules for updating dependencies are:

1. If X does not depend on another stream id, Y inherits the priority of X.
2. If X does depend on another stream id W, Y inherits the dependency pointer from X to W.

For example, for dependencies

```
a.htm <- a.js <- 1.png - 2.png
```

where the server receives an `END_STREAM_ACK` for 1.png, the resulting dependencies would be

```
a.htm <- a.js <- 2.png
```

Of course, clients may reconfigure dependencies using whatever policy they wish by sending an explicit `PRIORITY` frame for stream Y before the `END_STREAM_ACK` for stream X.

Updating dependencies when overwriting values is analogous to list insertion. If stream Y depends on X and a `HEADERS` or `PRIORITY` frame is received indicating a dependency on X for stream Z, Z replaces Y as X's dependent, and Y's dependency is updated to Z with the same ordering as it had to X. For example, if

```
a.htm - 1.png
```

and the server receives a HEADERS frame for a.js with an ordered dependency on a.htm, the result is

```
a.htm <- a.js - 1.png
```

5. Examples

The combination of dependencies and priorities suffices to express serialized as well as concurrent transfer schedules. But, how should the browser choose dependencies and priorities when making requests? This question is best answered quantitatively. As a starting point, we consider the following policy in our examples:

1. Resource dependencies reflect parser-blocking order. Non-streaming resources are serialized; i.e., non-async scripts and styling.
2. Progressive resources (e.g., images) are transferred concurrently and configured to depend on parser-blocking resource transfers.
3. To ensure that the speculative parser can maintain enough in-flight requests to fill the pipe between the client and server, page HTML does not depend on other streams. (Although, a background tab should have lower priority.)

Concretely, suppose a HTTP/2 connection is multiplexing multiple tabs from a user connected to a HTTP/2 proxy, with parent pointers and priorities as shown below. (P6, for example, indicates a priority of 6.)

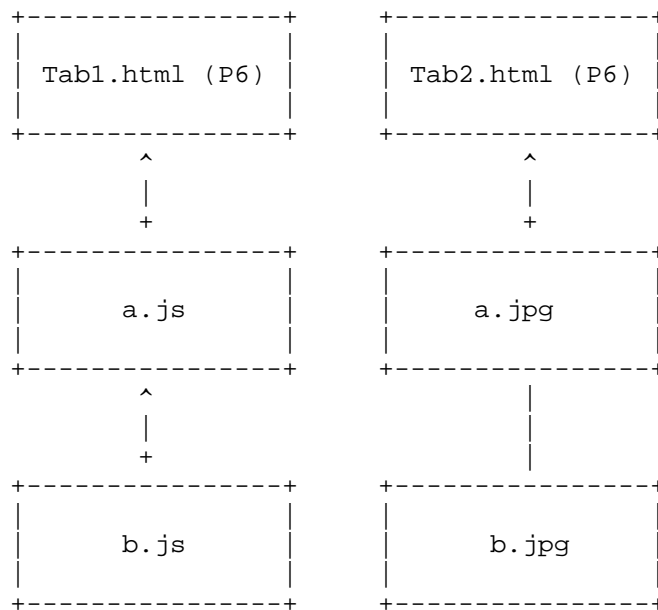


Figure 1: Multiple Tab Example

To color in this example, suppose that Tab 1 is the foreground tab, loading in parallel with Tab 2 in the background. Thus, its relatively higher weight. `a.js` and `b.js` are scripts required for the first tab and should be transferred serially (as scripts are executed in the order they are declared in the document, and are not parsed until transfer completes.) Thus, `a.js` depends on `b.js` depends on `tab1.htm`. In the background tab, two image transfers share capacity as both can be rendered progressively. Thus, the dependency between `b.jpg` and `a.jpg` is unordered, indicating that writes for the `tab2.html` stream should be scheduled first, but capacity may be shared between the streams for `a.jpg` and `b.jpg`.

When scheduling transfers, we consider a server that treats dependencies conceptually as lists. Recall that streams depend on and are depended on by at most one other stream. These can be treated as predecessor and successor ids. Stream writes are scheduled in two steps: 1) choosing a dependency list with at least one stream ready to write and 2) then selecting the stream to write by traversing the list. (An implementation might maintain ready queues of streams for efficiency, but we consider a simplified setting for clarity.)

Because the streams associated with the transfers of `tab1` and `tab2` have priorities rather than dependencies, they are always scheduled

before any dependent streams. But, bandwidth allocation between dependency lists remains proportional as defined by the relative priority of tab1 and tab2. For example, if the transfer of tab2.htm is in progress and tab1.htm (now complete) is ready and selected by the scheduler, a.js will be scheduled before tab2.htm completes. This process proceeds until all transfers in a list have completed.

5.1. Specifying an ordering of resource transfers and reacting to document parsing

We illustrate the need for both serial dependencies, concurrency, and reprioritization in these cases with a simple example.

Suppose site.com has index.htm:

```
<html>
<body>
<script src="a.js"></script>


</body>
```

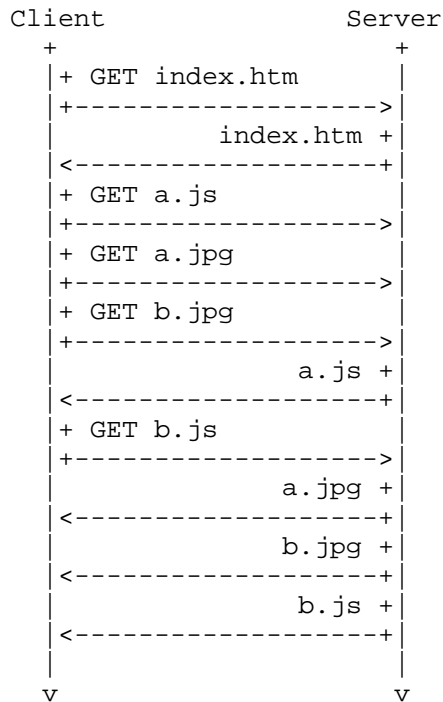
with a.js:

```
document.write('<script src="b.js"></script>');
```

and b.js:

```
document.write('<div>blocker</div>');
```

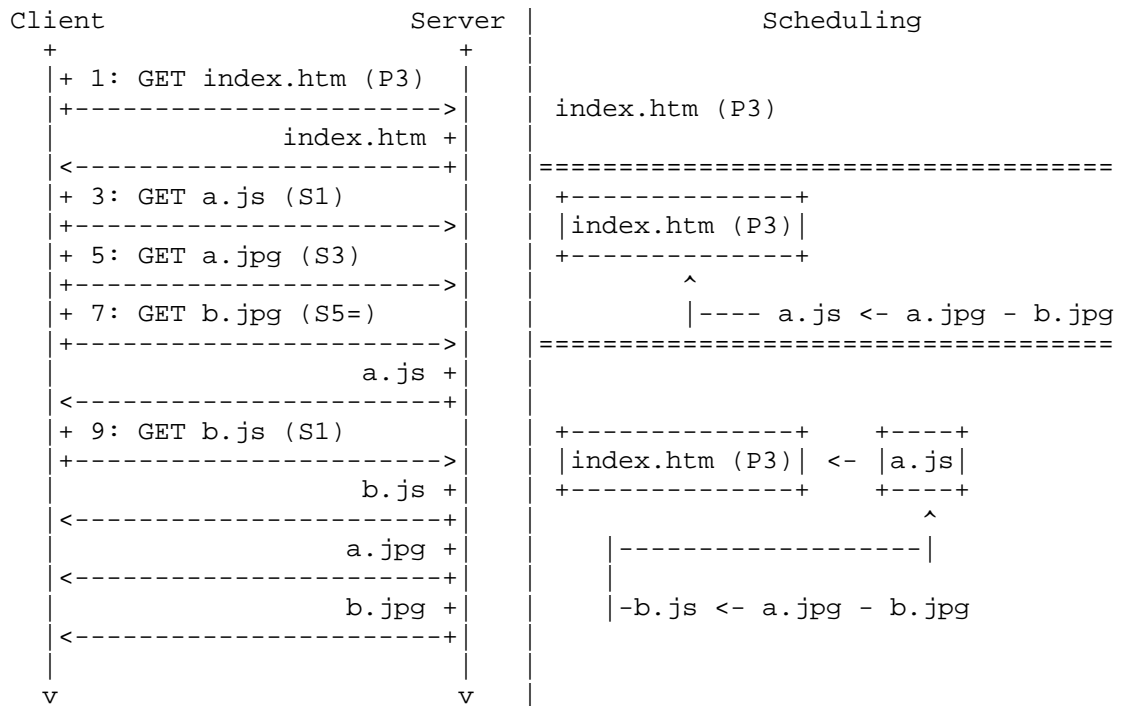
How would this example page be transferred today? As the main HTML is received and parsed, a request for a.js will be issued and block the document parser. As the remaining HTML streams in, the speculative parser will issue requests for a.jpg and b.jpg in quick succession. Once a.js is received and executed, a request for b.js will be issued, which again blocks parsing until received. Visually:



This transfer schedule is suboptimal. Page rendering will complete only when once b.js has completed, but receiving b.js is slowed by competition for bandwidth capacity for a.jpg and b.jpg, which do not block rendering.

Ideally, the order resources are transferred would reflect the document parse order with bandwidth sharing only for progressive resources. More specifically, we want to receive: 1) index.htm, 2) a.js, and 3) b.js sequentially. After those critical transfers have completed, a.jpg and b.jpg should be transferred concurrently since they may be displayed progressively.

Folding in the protocol mechanisms described above:



In the figure, each resource request corresponds to a new HTTP/2 stream with the form ID: request (PriOrDep). In more detail:

- o The HEADERS for the index.htm request indicates a default priority (3) and a stream id of 1.
- o The document parser is blocked once the external script a.js is parsed. At this point, the speculative parser looks ahead and creates new streams for a.jpg and b.jpg in parse order. a.jpg and b.jpg can be progressively rendered, so their transfer is concurrent (a.jpg has an ordered dependency on a.js, and b.jpg has an unordered dependency on a.jpg).
- o Once a.js completes, the document parser continues by executing a.js and inserting b.js via document.write(), again blocking document parsing on the receipt of b.js. At this point, b.js should preempt all other transfers since it's a non-streaming resource that is blocking page rendering. To this end, the client creates the b.js stream which depends on a.js (or, equivalently, index.htm).

This transfer schedule improves performance by serializing the transfer of resources on the critical path. The browser can ensure

that resources needed immediately do not compete for bandwidth capacity with less important transfers. The pipe remains full, as a queue of requests is maintained in the dependency list, filling any idle capacity with useful data. Where we cannot make an informed scheduling decision, we hedge our bets with concurrent transfers by hinting that they are unordered and letting the server decide what makes the most sense --- as in the case of two above the fold images that can be rendered progressively.

5.2. Servicing multiple tabs/users over a single HTTP/2 connection

As an illustration of this case, recall the example (Figure 1) from our straw-man design.

Suppose concurrent tabs are loading with the dependencies shown. When a user changes tabs, the browser sends a PRIORITY frame updating the stream associated with tab2.htm to, say, priority 8. (A batched message might also reduce the priority of tab1.htm to weight 3.) Because bandwidth is allocated among streams with priorities before considering their dependents, increasing the priority of tab2.htm effectively shifts capacity for all resource transfers depending on tab1.htm to tab2.htm.

5.3. Server Push

Push streams are assigned a priority or dependency at the discretion of the server. Typically, the Promised-Stream-ID would depend on the stream id carrying the PUSH_PROMISE frame. As information about resources needed for parsing is learned, the browser may update the dependency relationship by sending a PRIORITY message.

6. Policy Considerations

Both priorities and stream dependencies are advisory hints. Browsers may adopt sophisticated policies or leave dependencies entirely unspecified. Similarly, servers may incorporate dependency hints into very sophisticated schedulers or ignore them entirely. The protocol mechanisms for encoding dependencies are designed to be simple. But, these mechanisms afford a very flexible set of policies depending on how browsers and servers use them. This section expands on several policy considerations.

6.1. Assigning and updating dependencies

In our examples, we consider a browser that configures dependencies to reflect parser-blocking order for resources, updated as parsing continues. We expect this to improve performance, but browsers are

free to deviate from this policy, and there may be good reasons to do so. For example, if the parser-blocking order is highly dynamic (e.g., in response to many JS events), the overhead of updating dependencies may not be worth the cost, particularly for small transfers. A sophisticated client may base dependency update decisions on content-length and/or RTT, restricting updates to only those streams likely to benefit from it. Quantitative implementation experience is needed to determine how best to assign and update dependencies.

6.2. Server scheduling

A conformant server should respect the semantics of priorities and dependencies in its scheduling policy. Priorities indicate a preference for weighted scheduling (e.g., using a lottery scheduler [LOTTERYSCHEDULING]) among top-level streams; i.e., those created with a priority and not a dependency. Capacity should be shared among a sequence of streams with unordered dependencies.

Server scheduling should reflect guidance from dependencies, but it need not be strict. If all streams in a dependency tree have data available to write at the server, writes should be serviced first for top-level streams, then ordered dependents, with sharing among unordered streams. But, dependents that are ready to write should not starve to enforce a scheduling dependency. In other words, scheduling dependencies should not lead servers to waste capacity. If data is not available to continue writing the top-level stream, for example, a dependent ready to write should do so.

Finally, we point out that servers may improve performance even if clients do not provide dependency information or priorities. For example, an intelligent server may inspect the content type of resources to make informed prioritization decisions on its own without client guidance. (However, respecting client-provided hints when available is likely to improve performance, as clients have detailed knowledge of parser dependencies.)

6.3. Garbage collecting dependency information

HTTP/2 implementations must take care to protect themselves from the use of dependencies as a DoS vector. The protocol provides wide flexibility in this regard; servers are free to drop dependency or priority data at any time without sacrificing correctness.

Typically, we envision servers will drop dependency information along with other stream state when an `END_STREAM_ACK` frame is received or the session is closed.

7. Security Considerations

TODO

8. Informative References

[LOTTERYSCHEDULING]

Waldspurger, C. and W. Weihl, "Lottery scheduling: flexible proportional-share resource management", 1994, <<http://dl.acm.org/citation.cfm?id=1267639>>.

[PRELOADSCANNER]

Gentilcore, T., "The WebKit PreloadScanner", 2011, <<http://gent.ilcore.com/2011/01/webkit-preloadscanner.html>>.

Appendix A. Acknowledgements

This document resulted from discussions amongst the SPDY team at Google. The authors merely took that discussion and edited this document. The individuals who contributed to those discussions include, but are not limited to: Roberto Peon, Hasan Khalil, Ryan Hamilton, Jim Roskind, Bryan McQuade, Chris Bentzel, Ilya Grigorik.

Authors' Addresses

Michael Piatek
Google

Email: piatek@chromium.org

William Chan
Google

Email: willchan@chromium.org

HTTPbis Working Group
Internet-Draft
Intended status: Standards Track
Expires: August 21, 2015

R. Peon
Google, Inc
H. Ruellan
Canon CRF
February 17, 2015

HPACK - Header Compression for HTTP/2
draft-ietf-httpbis-header-compression-12

Abstract

This specification defines HPACK, a compression format for efficiently representing HTTP header fields, to be used in HTTP/2.

Editorial Note (To be removed by RFC Editor)

Discussion of this draft takes place on the HTTPBIS working group mailing list (ietf-http-wg@w3.org), which is archived at [1].

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

The changes in this draft are summarized in Appendix D.2.

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 August 21, 2015.

Copyright Notice

Copyright (c) 2015 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction	4
1.1. Overview	4
1.2. Conventions	5
1.3. Terminology	5
2. Compression Process Overview	6
2.1. Header List Ordering	6
2.2. Encoding and Decoding Contexts	6
2.3. Indexing Tables	6
2.3.1. Static Table	6
2.3.2. Dynamic Table	6
2.3.3. Index Address Space	7
2.4. Header Field Representation	8
3. Header Block Decoding	8
3.1. Header Block Processing	8
3.2. Header Field Representation Processing	9
4. Dynamic Table Management	9
4.1. Calculating Table Size	10
4.2. Maximum Table Size	10
4.3. Entry Eviction when Dynamic Table Size Changes	11
4.4. Entry Eviction when Adding New Entries	11
5. Primitive Type Representations	11
5.1. Integer Representation	11
5.2. String Literal Representation	13
6. Binary Format	14
6.1. Indexed Header Field Representation	14
6.2. Literal Header Field Representation	15
6.2.1. Literal Header Field with Incremental Indexing	15
6.2.2. Literal Header Field without Indexing	16
6.2.3. Literal Header Field never Indexed	17
6.3. Dynamic Table Size Update	18
7. Security Considerations	19
7.1. Probing Dynamic Table State	19
7.1.1. Applicability to HPACK and HTTP	20
7.1.2. Mitigation	20
7.1.3. Never Indexed Literals	21
7.2. Static Huffman Encoding	22

7.3. Memory Consumption	22
7.4. Implementation Limits	23
8. IANA Considerations	23
9. Acknowledgments	23
10. References	23
10.1. Normative References	23
10.2. Informative References	24
Appendix A. Static Table Definition	25
Appendix B. Huffman Code	26
Appendix C. Examples	32
C.1. Integer Representation Examples	33
C.1.1. Example 1: Encoding 10 Using a 5-bit Prefix	33
C.1.2. Example 2: Encoding 1337 Using a 5-bit Prefix	33
C.1.3. Example 3: Encoding 42 Starting at an Octet Boundary	34
C.2. Header Field Representation Examples	34
C.2.1. Literal Header Field with Indexing	34
C.2.2. Literal Header Field without Indexing	35
C.2.3. Literal Header Field never Indexed	36
C.2.4. Indexed Header Field	36
C.3. Request Examples without Huffman Coding	37
C.3.1. First Request	37
C.3.2. Second Request	38
C.3.3. Third Request	39
C.4. Request Examples with Huffman Coding	40
C.4.1. First Request	40
C.4.2. Second Request	41
C.4.3. Third Request	42
C.5. Response Examples without Huffman Coding	44
C.5.1. First Response	44
C.5.2. Second Response	46
C.5.3. Third Response	47
C.6. Response Examples with Huffman Coding	49
C.6.1. First Response	49
C.6.2. Second Response	51
C.6.3. Third Response	52
Appendix D. Change Log (to be removed by RFC Editor before publication)	54
D.1. Since draft-ietf-httpbis-header-compression-10	55
D.2. Since draft-ietf-httpbis-header-compression-09	55
D.3. Since draft-ietf-httpbis-header-compression-08	55
D.4. Since draft-ietf-httpbis-header-compression-07	55
D.5. Since draft-ietf-httpbis-header-compression-06	56
D.6. Since draft-ietf-httpbis-header-compression-05	56
D.7. Since draft-ietf-httpbis-header-compression-04	56
D.8. Since draft-ietf-httpbis-header-compression-03	57
D.9. Since draft-ietf-httpbis-header-compression-02	57
D.10. Since draft-ietf-httpbis-header-compression-01	57
D.11. Since draft-ietf-httpbis-header-compression-00	57

1. Introduction

In HTTP/1.1 (see [RFC7230]), header fields are not compressed. As Web pages have grown to require dozens to hundreds of requests, the redundant header fields in these requests unnecessarily consume bandwidth, measurably increasing latency.

SPDY [SPDY] initially addressed this redundancy by compressing header fields using the DEFLATE [DEFLATE] format, which proved very effective at efficiently representing the redundant header fields. However, that approach exposed a security risk as demonstrated by the CRIME attack (see [CRIME]).

This specification defines HPACK, a new compressor for header fields which eliminates redundant header fields, limits vulnerability to known security attacks, and which has a bounded memory requirement for use in constrained environments. Potential security concerns for HPACK are described in Section 7.

The HPACK format is intentionally simple and inflexible. Both characteristics reduce the risk of interoperability or security issues due to implementation error. No extensibility mechanisms are defined; changes to the format are only possible by defining a complete replacement.

1.1. Overview

The format defined in this specification treats a list of header fields as an ordered collection of name-value pairs that can include duplicate pairs. Names and values are considered to be opaque sequences of octets, and the order of header fields is preserved after being compressed and decompressed.

Encoding is informed by header field tables that map header fields to indexed values. These header field tables can be incrementally updated as new header fields are encoded or decoded.

In the encoded form, a header field is represented either literally or as a reference to a header field in one of the header field tables. Therefore, a list of header fields can be encoded using a mixture of references and literal values.

Literal values are either encoded directly or using a static Huffman code.

The encoder is responsible for deciding which header fields to insert as new entries in the header field tables. The decoder executes the modifications to the header field tables prescribed by the encoder,

reconstructing the list of header fields in the process. This enables decoders to remain simple and interoperate with a wide variety of encoders.

Examples illustrating the use of these different mechanisms to represent header fields are available in Appendix C.

1.2. Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

All numeric values are in network byte order. Values are unsigned unless otherwise indicated. Literal values are provided in decimal or hexadecimal as appropriate.

1.3. Terminology

This specification uses the following terms:

Header Field: A name-value pair. Both the name and value are treated as opaque sequences of octets.

Dynamic Table: The dynamic table (see Section 2.3.2) is a table that associates stored header fields with index values. This table is dynamic and specific to an encoding or decoding context.

Static Table: The static table (see Section 2.3.1) is a table that statically associates header fields that occur frequently with index values. This table is ordered, read-only, always accessible, and may be shared amongst all encoding or decoding contexts.

Header List: A header list is an ordered collection of header fields that are encoded jointly, and can contain duplicate header fields. A complete list of header fields contained in an HTTP/2 header block is a header list.

Header Field Representation: A header field can be represented in encoded form either as a literal or as an index (see Section 2.4).

Header Block: An ordered list of header field representations which, when decoded, yields a complete header list.

2. Compression Process Overview

This specification does not describe a specific algorithm for an encoder. Instead, it defines precisely how a decoder is expected to operate, allowing encoders to produce any encoding that this definition permits.

2.1. Header List Ordering

HPACK preserves the ordering of header fields inside the header list. An encoder **MUST** order header field representations in the header block according to their ordering in the original header list. A decoder **MUST** order header fields in the decoded header list according to their ordering in the header block.

2.2. Encoding and Decoding Contexts

To decompress header blocks, a decoder only needs to maintain a dynamic table (see Section 2.3.2) as a decoding context. No other dynamic state is needed.

When used for bidirectional communication, such as in HTTP, the encoding and decoding dynamic tables maintained by an endpoint are completely independent. I.e., the request and response dynamic tables are separate.

2.3. Indexing Tables

HPACK uses two tables for associating header fields to indexes. The static table (see Section 2.3.1) is predefined and contains common header fields (most of them with an empty value). The dynamic table (see Section 2.3.2) is dynamic and can be used by the encoder to index header fields repeated in the encoded header lists.

These two tables are combined into a single address space for defining index values (see Section 2.3.3).

2.3.1. Static Table

The static table consists of a predefined static list of header fields. Its entries are defined in Appendix A.

2.3.2. Dynamic Table

The dynamic table consists of a list of header fields maintained in first-in, first-out order. The first and newest entry in a dynamic table is at the lowest index, and the oldest entry of a dynamic table is at the highest index.

The dynamic table is initially empty. Entries are added as each header block is decompressed.

The dynamic table can contain duplicate entries (i.e., entries with the same name and same value). Therefore, duplicate entries **MUST NOT** be treated as an error by a decoder.

The encoder decides how to update the dynamic table and as such can control how much memory is used by the dynamic table. To limit the memory requirements of the decoder, the dynamic table size is strictly bounded (see Section 4.2).

The decoder updates the dynamic table during the processing of a list of header field representations (see Section 3.2).

2.3.3. Index Address Space

The static table and the dynamic table are combined into a single index address space.

Indices between 1 and the length of the static table (inclusive) refer to elements in the static table (see Section 2.3.1).

Indices strictly greater than the length of the static table refer to elements in the dynamic table (see Section 2.3.2). The length of the static table is subtracted to find the index into the dynamic table.

Indices strictly greater than the sum of the lengths of both tables **MUST** be treated as a decoding error.

For a static table size of s and a dynamic table size of k , the following diagram shows the entire valid index address space.

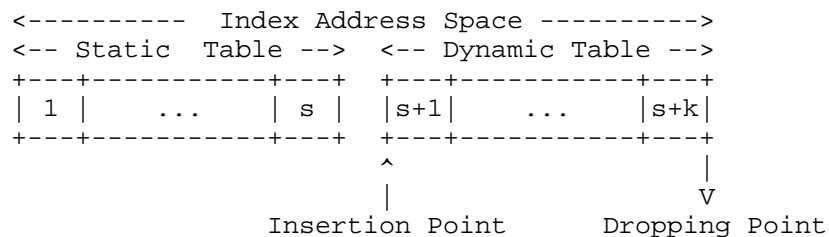


Figure 1: Index Address Space

2.4. Header Field Representation

An encoded header field can be represented either as an index or as a literal.

An indexed representation defines a header field as a reference to an entry in either the static table or the dynamic table (see Section 6.1).

A literal representation defines a header field by specifying its name and value. The header field name can be represented literally or as a reference to an entry in either the static table or the dynamic table. The header field value is represented literally.

Three different literal representations are defined:

- o A literal representation that adds the header field as a new entry at the beginning of the dynamic table (see Section 6.2.1).
- o A literal representation that does not add the header field to the dynamic table (see Section 6.2.2).
- o A literal representation that does not add the header field to the dynamic table, with the additional stipulation that this header field always use a literal representation, in particular when re-encoded by an intermediary (see Section 6.2.3). This representation is intended for protecting header field values that are not to be put at risk by compressing them (see Section 7.1.3 for more details).

The selection of one of these literal representations can be guided by security considerations, in order to protect sensitive header field values (see Section 7.1).

The literal representation of a header field name or of a header field value can encode the sequence of octets either directly or using a static Huffman code (see Section 5.2).

3. Header Block Decoding

3.1. Header Block Processing

A decoder processes a header block sequentially to reconstruct the original header list.

A header block is the concatenation of header field representations. The different possible header field representations are described in Section 6.

Once a header field is decoded and added to the reconstructed header list, the header field cannot be removed. A header field added to the header list can be safely passed to the application.

By passing the resulting header fields to the application, a decoder can be implemented with minimal transitory memory commitment in addition to the dynamic table.

3.2. Header Field Representation Processing

The processing of a header block to obtain a header list is defined in this section. To ensure that the decoding will successfully produce a header list, a decoder **MUST** obey the following rules.

All the header field representations contained in a header block are processed in the order in which they appear, as specified below. Details on the formatting of the various header field representations, and some additional processing instructions are found in Section 6.

An `_indexed representation_` entails the following actions:

- o The header field corresponding to the referenced entry in either the static table or dynamic table is appended to the decoded header list.

A `_literal representation_` that is `_not added_` to the dynamic table entails the following action:

- o The header field is appended to the decoded header list.

A `_literal representation_` that is `_added_` to the dynamic table entails the following actions:

- o The header field is appended to the decoded header list.
- o The header field is inserted at the beginning of the dynamic table. This insertion could result in the eviction of previous entries in the dynamic table (see Section 4.4).

4. Dynamic Table Management

To limit the memory requirements on the decoder side, the dynamic table is constrained in size.

4.1. Calculating Table Size

The size of the dynamic table is the sum of the size of its entries.

The size of an entry is the sum of its name's length in octets (as defined in Section 5.2), its value's length in octets, plus 32.

The size of an entry is calculated using the length of its name and value without any Huffman encoding applied.

Note: The additional 32 octets account for an estimated overhead associated with an entry. For example, an entry structure using two 64-bit pointers to reference the name and the value of the entry, and two 64-bit integers for counting the number of references to the name and value would have 32 octets of overhead.

4.2. Maximum Table Size

Protocols that use HPACK determine the maximum size that the encoder is permitted to use for the dynamic table. In HTTP/2, this value is determined by the `SETTINGS_HEADER_TABLE_SIZE` setting (see Section 6.5.2 of [HTTP2]).

An encoder can choose to use less capacity than this maximum size (see Section 6.3), but the chosen size **MUST** stay lower than or equal to the maximum set by the protocol.

A change in the maximum size of the dynamic table is signaled via an encoding context update (see Section 6.3). This encoding context update **MUST** occur at the beginning of the first header block following the change to the dynamic table size. In HTTP/2, this follows a settings acknowledgment (see Section 6.5.3 of [HTTP2]).

Multiple updates to the maximum table size can occur between the transmission of two header blocks. In the case that this size is changed more than once in this interval, the smallest maximum table size that occurs in that interval **MUST** be signaled in an encoding context update. The final maximum size is always signaled, resulting in at most two encoding context updates. This ensures that the decoder is able to perform eviction based on reductions in dynamic table size (see Section 4.3).

This mechanism can be used to completely clear entries from the dynamic table by setting a maximum size of 0, which can subsequently be restored.

4.3. Entry Eviction when Dynamic Table Size Changes

Whenever the maximum size for the dynamic table is reduced, entries are evicted from the end of the dynamic table until the size of the dynamic table is less than or equal to the maximum size.

4.4. Entry Eviction when Adding New Entries

Before a new entry is added to the dynamic table, entries are evicted from the end of the dynamic table until the size of the dynamic table is less than or equal to (maximum size - new entry size), or until the table is empty.

If the size of the new entry is less than or equal to the maximum size, that entry is added to the table. It is not an error to attempt to add an entry that is larger than the maximum size; an attempt to add an entry larger than the maximum size causes the table to be emptied of all existing entries, and results in an empty table.

A new entry can reference the name of an entry in the dynamic table that will be evicted when adding this new entry into the dynamic table. Implementations are cautioned to avoid deleting the referenced name if the referenced entry is evicted from the dynamic table prior to inserting the new entry.

5. Primitive Type Representations

HPACK encoding uses two primitive types: unsigned variable length integers, and strings of octets.

5.1. Integer Representation

Integers are used to represent name indexes, header field indexes or string lengths. An integer representation can start anywhere within an octet. To allow for optimized processing, an integer representation always finishes at the end of an octet.

An integer is represented in two parts: a prefix that fills the current octet and an optional list of octets that are used if the integer value does not fit within the prefix. The number of bits of the prefix (called N) is a parameter of the integer representation.

If the integer value is small enough, i.e., strictly less than $2^N - 1$, it is encoded within the N -bit prefix.

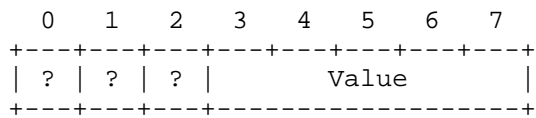


Figure 2: Integer Value Encoded within the Prefix (shown for N = 5)

Otherwise, all the bits of the prefix are set to 1 and the value, decreased by 2^N-1 , is encoded using a list of one or more octets. The most significant bit of each octet is used as a continuation flag: its value is set to 1 except for the last octet in the list. The remaining bits of the octets are used to encode the decreased value.

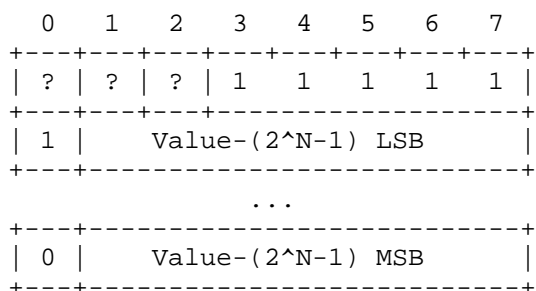


Figure 3: Integer Value Encoded after the Prefix (shown for N = 5)

Decoding the integer value from the list of octets starts by reversing the order of the octets in the list. Then, for each octet, its most significant bit is removed. The remaining bits of the octets are concatenated and the resulting value is increased by 2^N-1 to obtain the integer value.

The prefix size, N, is always between 1 and 8 bits. An integer starting at an octet-boundary will have an 8-bit prefix.

Pseudo-code to represent an integer I is as follows:

```

if I <  $2^N - 1$ , encode I on N bits
else
  encode ( $2^N - 1$ ) on N bits
  I = I - ( $2^N - 1$ )
  while I >= 128
    encode (I % 128 + 128) on 8 bits
    I = I / 128
  encode I on 8 bits

```

Pseudo-code to decode an integer I is as follows:

```

decode I from the next N bits
if I < 2^N - 1, return I
else
    M = 0
    repeat
        B = next octet
        I = I + (B & 127) * 2^M
        M = M + 7
    while B & 128 == 128
    return I

```

Examples illustrating the encoding of integers are available in Appendix C.1.

This integer representation allows for values of indefinite size. It is also possible for an encoder to send a large number of zero values, which can waste octets and could be used to overflow integer values. Integer encodings that exceed an implementation limits - in value or octet length - MUST be treated as a decoding error. Different limits can be set for each of the different uses of integers, based on implementation constraints.

5.2. String Literal Representation

Header field names and header field values can be represented as literal strings. A literal string is encoded as a sequence of octets, either by directly encoding the literal string's octets, or by using a Huffman code (see [HUFFMAN]).

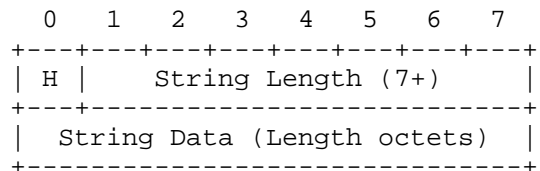


Figure 4: String Literal Representation

A literal string representation contains the following fields:

H: A one bit flag, H, indicating whether or not the octets of the string are Huffman encoded.

String Length: The number of octets used to encode the string literal, encoded as an integer with 7-bit prefix (see Section 5.1).

String Data: The encoded data of the string literal. If H is '0', then the encoded data is the raw octets of the string literal. If H is '1', then the encoded data is the Huffman encoding of the string literal.

String literals which use Huffman encoding are encoded with the Huffman code defined in Appendix B (see examples for requests in Appendix C.4 and for responses in Appendix C.6). The encoded data is the bitwise concatenation of the codes corresponding to each octet of the string literal.

As the Huffman encoded data doesn't always end at an octet boundary, some padding is inserted after it, up to the next octet boundary. To prevent this padding to be misinterpreted as part of the string literal, the most significant bits of the code corresponding to the EOS (end-of-string) symbol are used.

Upon decoding, an incomplete code at the end of the encoded data is to be considered as padding and discarded. A padding strictly longer than 7 bits MUST be treated as a decoding error. A padding not corresponding to the most significant bits of the code for the EOS symbol MUST be treated as a decoding error. A Huffman encoded string literal containing the EOS symbol MUST be treated as a decoding error.

6. Binary Format

This section describes the detailed format of each of the different header field representations, plus the encoding context update instruction.

6.1. Indexed Header Field Representation

An indexed header field representation identifies an entry in either the static table or the dynamic table (see Section 2.3).

An indexed header field representation causes a header field to be added to the decoded header list, as described in Section 3.2.

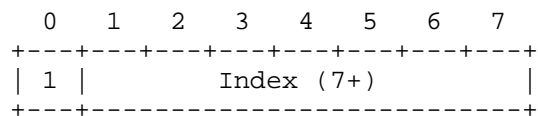


Figure 5: Indexed Header Field

An indexed header field starts with the '1' 1-bit pattern, followed by the index of the matching header field, represented as an integer with a 7-bit prefix (see Section 5.1).

The index value of 0 is not used. It MUST be treated as a decoding error if found in an indexed header field representation.

6.2. Literal Header Field Representation

A literal header field representation contains a literal header field value. Header field names are either provided as a literal or by reference to an existing table entry, either from the static table or the dynamic table (see Section 2.3).

This specification defines three forms of literal header field representations; with indexing, without indexing, and never indexed.

6.2.1. Literal Header Field with Incremental Indexing

A literal header field with incremental indexing representation results in appending a header field to the decoded header list and inserting it as a new entry into the dynamic table.

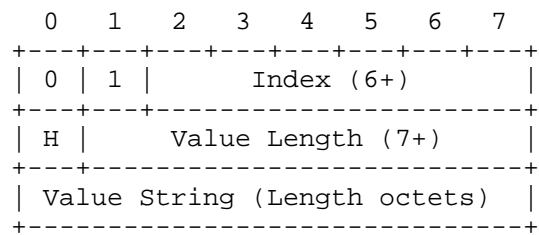


Figure 6: Literal Header Field with Incremental Indexing - Indexed Name

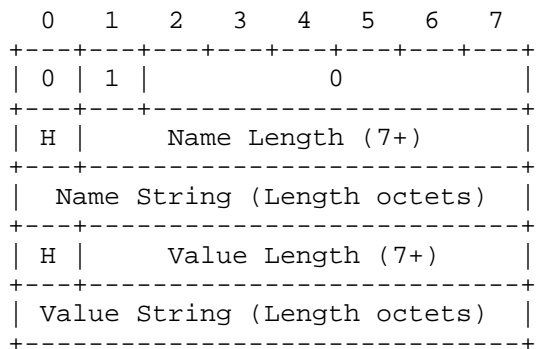


Figure 7: Literal Header Field with Incremental Indexing - New Name

A literal header field with incremental indexing representation starts with the '01' 2-bit pattern.

If the header field name matches the header field name of an entry stored in the static table or the dynamic table, the header field name can be represented using the index of that entry. In this case, the index of the entry is represented as an integer with a 6-bit prefix (see Section 5.1). This value is always non-zero.

Otherwise, the header field name is represented as a literal string (see Section 5.2). A value 0 is used in place of the 6-bit index, followed by the header field name.

Either form of header field name representation is followed by the header field value represented as a literal string (see Section 5.2).

6.2.2. Literal Header Field without Indexing

A literal header field without indexing representation results in appending a header field to the decoded header list without altering the dynamic table.

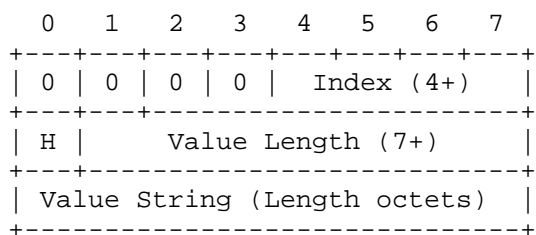


Figure 8: Literal Header Field without Indexing - Indexed Name

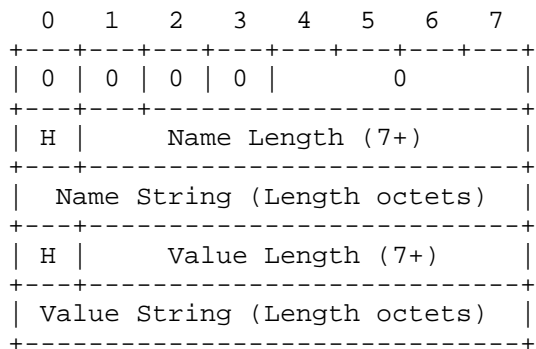


Figure 9: Literal Header Field without Indexing - New Name

A literal header field without indexing representation starts with the '0000' 4-bit pattern.

If the header field name matches the header field name of an entry stored in the static table or the dynamic table, the header field name can be represented using the index of that entry. In this case, the index of the entry is represented as an integer with a 4-bit prefix (see Section 5.1). This value is always non-zero.

Otherwise, the header field name is represented as a literal string (see Section 5.2). A value 0 is used in place of the 4-bit index, followed by the header field name.

Either form of header field name representation is followed by the header field value represented as a literal string (see Section 5.2).

6.2.3. Literal Header Field never Indexed

A literal header field never indexed representation results in appending a header field to the decoded header list without altering the dynamic table. Intermediaries MUST use the same representation for encoding this header field.

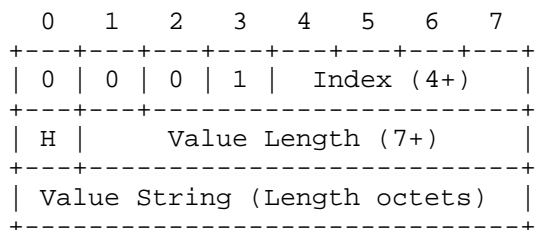


Figure 10: Literal Header Field never Indexed - Indexed Name

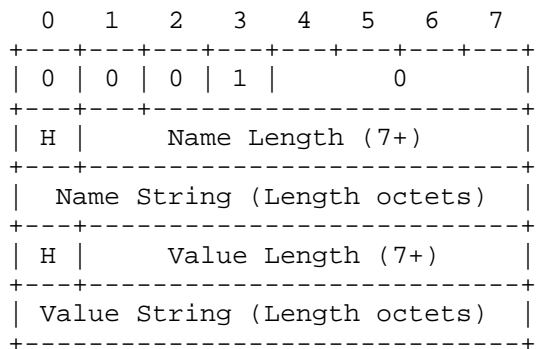


Figure 11: Literal Header Field never Indexed - New Name

A literal header field never indexed representation starts with the '0001' 4-bit pattern.

When a header field is represented as a literal header field never indexed, it **MUST** always be encoded with this specific literal representation. In particular, when a peer sends a header field that it received represented as a literal header field never indexed, it **MUST** use the same representation to forward this header field.

This representation is intended for protecting header field values that are not to be put at risk by compressing them (see Section 7.1 for more details).

The encoding of the representation is identical to the literal header field without indexing (see Section 6.2.2).

6.3. Dynamic Table Size Update

A dynamic table size update signals a change to the size of the dynamic table.

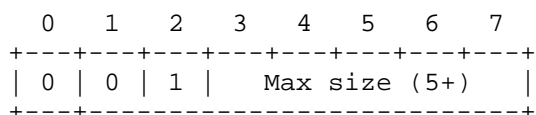


Figure 12: Maximum Dynamic Table Size Change

A dynamic table size update starts with the '001' 3-bit pattern, followed by the new maximum size, represented as an integer with a 5-bit prefix (see Section 5.1).

The new maximum size MUST be lower than or equal to the last value of the maximum size of the dynamic table. A value that exceeds this limit MUST be treated as a decoding error. In HTTP/2, this limit is the last value of the `SETTINGS_HEADER_TABLE_SIZE` parameter (see Section 6.5.2 of [HTTP2]) received from the decoder and acknowledged by the encoder (see Section 6.5.3 of [HTTP2]).

Reducing the maximum size of the dynamic table can cause entries to be evicted (see Section 4.3).

7. Security Considerations

This section describes potential areas of security concern with HPACK:

- o Use of compression as a length-based oracle for verifying guesses about secrets that are compressed into a shared compression context.
- o Denial of service resulting from exhausting processing or memory capacity at a decoder.

7.1. Probing Dynamic Table State

HPACK reduces the length of header field encodings by exploiting the redundancy inherent in protocols like HTTP. The ultimate goal of this is to reduce the amount of data that is required to send HTTP requests or responses.

The compression context used to encode header fields can be probed by an attacker who can both define header fields to be encoded and transmitted and observe the length of those fields once they are encoded. When an attacker can do both, they can adaptively modify requests in order to confirm guesses about the dynamic table state. If a guess is compressed into a shorter length, the attacker can observe the encoded length and infer that the guess was correct.

This is possible even over the Transport Layer Security Protocol (TLS, see [TLS12]), because while TLS provides confidentiality protection for content, it only provides a limited amount of protection for the length of that content.

Note: Padding schemes only provide limited protection against an attacker with these capabilities, potentially only forcing an increased number of guesses to learn the length associated with a given guess. Padding schemes also work directly against compression by increasing the number of bits that are transmitted.

Attacks like CRIME [CRIME] demonstrated the existence of these general attacker capabilities. The specific attack exploited the fact that DEFLATE [DEFLATE] removes redundancy based on prefix matching. This permitted the attacker to confirm guesses a character at a time, reducing an exponential-time attack into a linear-time attack.

7.1.1. Applicability to HPACK and HTTP

HPACK mitigates but does not completely prevent attacks modeled on CRIME [CRIME] by forcing a guess to match an entire header field value, rather than individual characters. An attacker can only learn whether a guess is correct or not, so is reduced to a brute force guess for the header field values.

The viability of recovering specific header field values therefore depends on the entropy of values. As a result, values with high entropy are unlikely to be recovered successfully. However, values with low entropy remain vulnerable.

Attacks of this nature are possible any time that two mutually distrustful entities control requests or responses that are placed onto a single HTTP/2 connection. If the shared HPACK compressor permits one entity to add entries to the dynamic table, and the other to access those entries, then the state of the table can be learned.

Having requests or responses from mutually distrustful entities occurs when an intermediary either:

- o sends requests from multiple clients on a single connection toward an origin server, or
- o takes responses from multiple origin servers and places them on a shared connection toward a client.

Web browsers also need to assume that requests made on the same connection by different web origins [ORIGIN] are made by mutually distrustful entities.

7.1.2. Mitigation

Users of HTTP that require confidentiality for header fields can use values with entropy sufficient to make guessing infeasible. However, this is impractical as a general solution because it forces all users of HTTP to take steps to mitigate attacks. It would impose new constraints on how HTTP is used.

Rather than impose constraints on users of HTTP, an implementation of HPACK can instead constrain how compression is applied in order to limit the potential for dynamic table probing.

An ideal solution segregates access to the dynamic table based on the entity that is constructing header fields. Header field values that are added to the table are attributed to an entity, and only the entity that created a particular value can extract that value.

To improve compression performance of this option, certain entries might be tagged as being public. For example, a web browser might make the values of the Accept-Encoding header field available in all requests.

An encoder without good knowledge of the provenance of header fields might instead introduce a penalty for a header field with many different values, such that a large number of attempts to guess a header field value results in the header field no more being compared to the dynamic table entries in future messages, effectively preventing further guesses.

Note: Simply removing entries corresponding to the header field from the dynamic table can be ineffectual if the attacker has a reliable way of causing values to be reinstalled. For example, a request to load an image in a web browser typically includes the Cookie header field (a potentially highly valued target for this sort of attack), and web sites can easily force an image to be loaded, thereby refreshing the entry in the dynamic table.

This response might be made inversely proportional to the length of the header field value. Marking a header field as not using the dynamic table any more might occur for shorter values more quickly or with higher probability than for longer values.

7.1.3. Never Indexed Literals

Implementations can also choose to protect sensitive header fields by not compressing them and instead encoding their value as literals.

Refusing to generate an indexed representation for a header field is only effective if compression is avoided on all hops. The never indexed literal (see Section 6.2.3) can be used to signal to intermediaries that a particular value was intentionally sent as a literal.

An intermediary **MUST NOT** re-encode a value that uses the never indexed literal representation with another representation that would

index it. If HPACK is used for re-encoding, the never indexed literal representation MUST be used.

The choice to use a never indexed literal representation for a header field depends on several factors. Since HPACK doesn't protect against guessing an entire header field value, short or low-entropy values are more readily recovered by an adversary. Therefore, an encoder might choose not to index values with low entropy.

An encoder might also choose not to index values for header fields that are considered to be highly valuable or sensitive to recovery, such as the Cookie or Authorization header fields.

On the contrary, an encoder might prefer indexing values for header fields that have little or no value if they were exposed. For instance, a User-Agent header field does not commonly vary between requests and is sent to any server. In that case, confirmation that a particular User-Agent value has been used provides little value.

Note that these criteria for deciding to use a never indexed literal representation will evolve over time as new attacks are discovered.

7.2. Static Huffman Encoding

There is no currently known attack against a static Huffman encoding. A study has shown that using a static Huffman encoding table created an information leakage, however this same study concluded that an attacker could not take advantage of this information leakage to recover any meaningful amount of information (see [PETAL]).

7.3. Memory Consumption

An attacker can try to cause an endpoint to exhaust its memory. HPACK is designed to limit both the peak and state amounts of memory allocated by an endpoint.

The amount of memory used by the compressor is limited by the protocol using HPACK through the definition of the maximum size of the dynamic table. In HTTP/2, this value is controlled by the decoder through the setting parameter SETTINGS_HEADER_TABLE_SIZE (see Section 6.5.2 of [HTTP2]). This limit takes into account both the size of the data stored in the dynamic table, plus a small allowance for overhead.

A decoder can limit the amount of state memory used by setting an appropriate value for the maximum size of the dynamic table. In HTTP/2, this is realized by setting an appropriate value for the SETTINGS_HEADER_TABLE_SIZE parameter. An encoder can limit the

amount of state memory it uses by signaling lower dynamic table size than the decoder allows (see Section 6.3).

The amount of temporary memory consumed by an encoder or decoder can be limited by processing header fields sequentially. An implementation does not need to retain a complete list of header fields. Note however that it might be necessary for an application to retain a complete header list for other reasons; even though HPACK does not force this to occur, application constraints might make this necessary.

7.4. Implementation Limits

An implementation of HPACK needs to ensure that large values for integers, long encoding for integers, or long string literals do not create security weaknesses.

An implementation has to set a limit for the values it accepts for integers, as well as for the encoded length (see Section 5.1). In the same way, it has to set a limit to the length it accepts for string literals (see Section 5.2).

8. IANA Considerations

This document has no IANA actions.

9. Acknowledgments

This specification includes substantial input from the following individuals:

- o Mike Bishop, Jeff Pinner, Julian Reschke, Martin Thomson (substantial editorial contributions).
- o Johnny Graettinger (Huffman code statistics).

10. References

10.1. Normative References

- [HTTP2] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol version 2", draft-ietf-httpbis-http2-17 (work in progress), February 2015.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

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

10.2. Informative References

- [CANONICAL] Schwartz, E. and B. Kallick, "Generating a canonical prefix encoding", Communications of the ACM Volume 7 Issue 3, pp. 166-169, March 1964, <<https://dl.acm.org/citation.cfm?id=363991>>.
- [CRIME] Rizzo, J. and T. Duong, "The CRIME Attack", September 2012, <https://docs.google.com/a/twist.com/presentation/d/11eBmGiHbYcHR9gL5nDyZChu_lCa2GizeuOfaLU2HOU>.
- [DEFLATE] Deutsch, P., "DEFLATE Compressed Data Format Specification version 1.3", RFC 1951, May 1996.
- [HUFFMAN] Huffman, D., "A Method for the Construction of Minimum Redundancy Codes", Proceedings of the Institute of Radio Engineers Volume 40, Number 9, pp. 1098-1101, September 1952, <<http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4051119>>.
- [ORIGIN] Barth, A., "The Web Origin Concept", RFC 6454, December 2011.
- [PETAL] Tan, J. and J. Nahata, "PETAL: Preset Encoding Table Information Leakage", April 2013, <<http://www.pdl.cmu.edu/PDL-FTP/associated/CMU-PDL-13-106.pdf>>.
- [SPDY] Belshe, M. and R. Peon, "SPDY Protocol", draft-mbelshe-httpbis-spdy-00 (work in progress), February 2012.
- [TLS12] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

Appendix A. Static Table Definition

The static table (see Section 2.3.1) consists in a predefined and unchangeable list of header fields.

The static table was created from the most frequent header fields used by popular web sites, with the addition of HTTP/2-specific pseudo-header fields (see Section 8.1.2.1 of [HTTP2]). For header fields with a few frequent values, an entry was added for each of these frequent values. For other header fields, an entry was added with an empty value.

The following table lists the predefined header fields that make-up the static table.

Index	Header Name	Header Value
1	:authority	
2	:method	GET
3	:method	POST
4	:path	/
5	:path	/index.html
6	:scheme	http
7	:scheme	https
8	:status	200
9	:status	204
10	:status	206
11	:status	304
12	:status	400
13	:status	404
14	:status	500
15	accept-charset	
16	accept-encoding	gzip, deflate
17	accept-language	
18	accept-ranges	
19	accept	
20	access-control-allow-origin	
21	age	
22	allow	
23	authorization	
24	cache-control	
25	content-disposition	
26	content-encoding	
27	content-language	
28	content-length	
29	content-location	
30	content-range	

31	content-type
32	cookie
33	date
34	etag
35	expect
36	expires
37	from
38	host
39	if-match
40	if-modified-since
41	if-none-match
42	if-range
43	if-unmodified-since
44	last-modified
45	link
46	location
47	max-forwards
48	proxy-authenticate
49	proxy-authorization
50	range
51	referer
52	refresh
53	retry-after
54	server
55	set-cookie
56	strict-transport-security
57	transfer-encoding
58	user-agent
59	vary
60	via
61	www-authenticate

Table 1: Static Table Entries

Table 1 gives the index of each entry in the static table.

Appendix B. Huffman Code

The following Huffman code is used when encoding string literals with a Huffman coding (see Section 5.2).

This Huffman code was generated from statistics obtained on a large sample of HTTP headers. It is a canonical Huffman code (see [CANONICAL]) with some tweaking to ensure that no symbol has a unique code length.

Each row in the table defines the code used to represent a symbol:

sym: The symbol to be represented. It is the decimal value of an octet, possibly prepended with its ASCII representation. A specific symbol, "EOS", is used to indicate the end of a string literal.

code as bits: The Huffman code for the symbol represented as a base-2 integer, aligned on the most significant bit (MSB).

code as hex: The Huffman code for the symbol, represented as a hexadecimal integer, aligned on the least significant bit (LSB).

len: The number of bits for the code representing the symbol.

As an example, the code for the symbol 47 (corresponding to the ASCII character "/") consists in the 6 bits "0", "1", "1", "0", "0", "0". This corresponds to the value 0x18 (in hexadecimal) encoded in 6 bits.

sym	code as bits aligned to MSB				code as hex aligned to LSB	len in bits
(0)	11111111	11000			1ff8	[13]
(1)	11111111	11111111	1011000		7fffd8	[23]
(2)	11111111	11111111	11111110	0010	fffffe2	[28]
(3)	11111111	11111111	11111110	0011	fffffe3	[28]
(4)	11111111	11111111	11111110	0100	fffffe4	[28]
(5)	11111111	11111111	11111110	0101	fffffe5	[28]
(6)	11111111	11111111	11111110	0110	fffffe6	[28]
(7)	11111111	11111111	11111110	0111	fffffe7	[28]
(8)	11111111	11111111	11111110	1000	fffffe8	[28]
(9)	11111111	11111111	11101010		ffffea	[24]
(10)	11111111	11111111	11111111	111100	3ffffffc	[30]
(11)	11111111	11111111	11111110	1001	fffffe9	[28]
(12)	11111111	11111111	11111110	1010	fffffea	[28]
(13)	11111111	11111111	11111111	111101	3ffffffd	[30]
(14)	11111111	11111111	11111110	1011	fffffeb	[28]
(15)	11111111	11111111	11111110	1100	fffffec	[28]
(16)	11111111	11111111	11111110	1101	fffffed	[28]
(17)	11111111	11111111	11111110	1110	fffffee	[28]
(18)	11111111	11111111	11111110	1111	fffffef	[28]
(19)	11111111	11111111	11111111	0000	fffff0	[28]
(20)	11111111	11111111	11111111	0001	fffff1	[28]
(21)	11111111	11111111	11111111	0010	fffff2	[28]
(22)	11111111	11111111	11111111	111110	3ffffffe	[30]
(23)	11111111	11111111	11111111	0011	fffff3	[28]
(24)	11111111	11111111	11111111	0100	fffff4	[28]
(25)	11111111	11111111	11111111	0101	fffff5	[28]

(26)	11111111 11111111 11111111 0110	ffffff6 [28]
(27)	11111111 11111111 11111111 0111	ffffff7 [28]
(28)	11111111 11111111 11111111 1000	ffffff8 [28]
(29)	11111111 11111111 11111111 1001	ffffff9 [28]
(30)	11111111 11111111 11111111 1010	ffffffa [28]
(31)	11111111 11111111 11111111 1011	ffffffb [28]
' ' (32)	010100	14 [6]
'!' (33)	11111110 00	3f8 [10]
'"' (34)	11111110 01	3f9 [10]
'#' (35)	11111111 1010	ffa [12]
'\$' (36)	11111111 11001	1ff9 [13]
'%' (37)	010101	15 [6]
'&' (38)	11111000	f8 [8]
' ' (39)	11111111 010	7fa [11]
'(' (40)	11111110 10	3fa [10]
')' (41)	11111110 11	3fb [10]
'*' (42)	11111001	f9 [8]
'+' (43)	11111111 011	7fb [11]
',' (44)	11111010	fa [8]
'-' (45)	010110	16 [6]
'.' (46)	010111	17 [6]
'/' (47)	011000	18 [6]
'0' (48)	00000	0 [5]
'1' (49)	00001	1 [5]
'2' (50)	00010	2 [5]
'3' (51)	011001	19 [6]
'4' (52)	011010	1a [6]
'5' (53)	011011	1b [6]
'6' (54)	011100	1c [6]
'7' (55)	011101	1d [6]
'8' (56)	011110	1e [6]
'9' (57)	011111	1f [6]
':' (58)	1011100	5c [7]
';' (59)	11111011	fb [8]
'<' (60)	11111111 1111100	7ffc [15]
'=' (61)	100000	20 [6]
'>' (62)	11111111 1011	ffb [12]
'?' (63)	11111111 00	3fc [10]
'@' (64)	11111111 11010	1ffa [13]
'A' (65)	100001	21 [6]
'B' (66)	1011101	5d [7]
'C' (67)	1011110	5e [7]
'D' (68)	1011111	5f [7]
'E' (69)	1100000	60 [7]
'F' (70)	1100001	61 [7]
'G' (71)	1100010	62 [7]
'H' (72)	1100011	63 [7]
'I' (73)	1100100	64 [7]

'J' (74)	1100101	65 [7]
'K' (75)	1100110	66 [7]
'L' (76)	1100111	67 [7]
'M' (77)	1101000	68 [7]
'N' (78)	1101001	69 [7]
'O' (79)	1101010	6a [7]
'P' (80)	1101011	6b [7]
'Q' (81)	1101100	6c [7]
'R' (82)	1101101	6d [7]
'S' (83)	1101110	6e [7]
'T' (84)	1101111	6f [7]
'U' (85)	1110000	70 [7]
'V' (86)	1110001	71 [7]
'W' (87)	1110010	72 [7]
'X' (88)	11111100	fc [8]
'Y' (89)	1110011	73 [7]
'Z' (90)	11111101	fd [8]
'[' (91)	11111111 11011	1fffb [13]
'\' (92)	11111111 11111110 000	7fff0 [19]
']' (93)	11111111 11100	1fffc [13]
'^' (94)	11111111 111100	3fffc [14]
'_' (95)	100010	22 [6]
'`' (96)	11111111 1111101	7fffd [15]
'a' (97)	00011	3 [5]
'b' (98)	100011	23 [6]
'c' (99)	00100	4 [5]
'd' (100)	100100	24 [6]
'e' (101)	00101	5 [5]
'f' (102)	100101	25 [6]
'g' (103)	100110	26 [6]
'h' (104)	100111	27 [6]
'i' (105)	00110	6 [5]
'j' (106)	1110100	74 [7]
'k' (107)	1110101	75 [7]
'l' (108)	101000	28 [6]
'm' (109)	101001	29 [6]
'n' (110)	101010	2a [6]
'o' (111)	00111	7 [5]
'p' (112)	101011	2b [6]
'q' (113)	1110110	76 [7]
'r' (114)	101100	2c [6]
's' (115)	01000	8 [5]
't' (116)	01001	9 [5]
'u' (117)	101101	2d [6]
'v' (118)	1110111	77 [7]
'w' (119)	1111000	78 [7]
'x' (120)	1111001	79 [7]
'y' (121)	1111010	7a [7]

'z' (122)	1111011			7b [7]
'{' (123)	11111111 1111110			7ffe [15]
' ' (124)	11111111 100			7fc [11]
'}' (125)	11111111 111101			3ffd [14]
'~' (126)	11111111 11101			1ffd [13]
(127)	11111111 11111111 11111111 1100			ffffffc [28]
(128)	11111111 11111110 0110			fffe6 [20]
(129)	11111111 11111111 010010			3ffd2 [22]
(130)	11111111 11111110 0111			fffe7 [20]
(131)	11111111 11111110 1000			fffe8 [20]
(132)	11111111 11111111 010011			3ffd3 [22]
(133)	11111111 11111111 010100			3ffd4 [22]
(134)	11111111 11111111 010101			3ffd5 [22]
(135)	11111111 11111111 1011001			7ffd9 [23]
(136)	11111111 11111111 010110			3ffd6 [22]
(137)	11111111 11111111 1011010			7ffdda [23]
(138)	11111111 11111111 1011011			7ffddb [23]
(139)	11111111 11111111 1011100			7fffdc [23]
(140)	11111111 11111111 1011101			7ffdd [23]
(141)	11111111 11111111 1011110			7ffdde [23]
(142)	11111111 11111111 11101011			ffffeb [24]
(143)	11111111 11111111 1011111			7ffddf [23]
(144)	11111111 11111111 11101100			ffffec [24]
(145)	11111111 11111111 11101101			ffffed [24]
(146)	11111111 11111111 010111			3ffd7 [22]
(147)	11111111 11111111 1100000			7ffe0 [23]
(148)	11111111 11111111 11101110			ffffee [24]
(149)	11111111 11111111 1100001			7ffe1 [23]
(150)	11111111 11111111 1100010			7ffe2 [23]
(151)	11111111 11111111 1100011			7ffe3 [23]
(152)	11111111 11111111 1100100			7ffe4 [23]
(153)	11111111 11111110 11100			1fffdc [21]
(154)	11111111 11111111 011000			3ffd8 [22]
(155)	11111111 11111111 1100101			7ffe5 [23]
(156)	11111111 11111111 011001			3ffd9 [22]
(157)	11111111 11111111 1100110			7ffe6 [23]
(158)	11111111 11111111 1100111			7ffe7 [23]
(159)	11111111 11111111 11101111			ffffef [24]
(160)	11111111 11111111 011010			3ffdda [22]
(161)	11111111 11111110 11101			1ffdd [21]
(162)	11111111 11111110 1001			fffe9 [20]
(163)	11111111 11111111 011011			3ffddb [22]
(164)	11111111 11111111 011100			3fffdc [22]
(165)	11111111 11111111 1101000			7ffe8 [23]
(166)	11111111 11111111 1101001			7ffe9 [23]
(167)	11111111 11111110 11110			1ffde [21]
(168)	11111111 11111111 1101010			7fffea [23]
(169)	11111111 11111111 011101			3ffdd [22]

(170)	11111111 11111111 011110	3ffffde [22]
(171)	11111111 11111111 11110000	ffffff0 [24]
(172)	11111111 11111110 11111	1fffdff [21]
(173)	11111111 11111111 011111	3fffdff [22]
(174)	11111111 11111111 1101011	7ffffeb [23]
(175)	11111111 11111111 1101100	7ffffec [23]
(176)	11111111 11111111 00000	1ffffe0 [21]
(177)	11111111 11111111 00001	1ffffe1 [21]
(178)	11111111 11111111 100000	3ffffe0 [22]
(179)	11111111 11111111 00010	1ffffe2 [21]
(180)	11111111 11111111 1101101	7ffffed [23]
(181)	11111111 11111111 100001	3ffffe1 [22]
(182)	11111111 11111111 1101110	7ffffee [23]
(183)	11111111 11111111 1101111	7ffffef [23]
(184)	11111111 11111110 1010	fffea [20]
(185)	11111111 11111111 100010	3ffffe2 [22]
(186)	11111111 11111111 100011	3ffffe3 [22]
(187)	11111111 11111111 100100	3ffffe4 [22]
(188)	11111111 11111111 1110000	7fffff0 [23]
(189)	11111111 11111111 100101	3ffffe5 [22]
(190)	11111111 11111111 100110	3ffffe6 [22]
(191)	11111111 11111111 1110001	7fffff1 [23]
(192)	11111111 11111111 11111000 00	3ffffe0 [26]
(193)	11111111 11111111 11111000 01	3ffffe1 [26]
(194)	11111111 11111110 1011	fffeb [20]
(195)	11111111 11111110 001	7ffff1 [19]
(196)	11111111 11111111 100111	3ffffe7 [22]
(197)	11111111 11111111 1110010	7fffff2 [23]
(198)	11111111 11111111 101000	3ffffe8 [22]
(199)	11111111 11111111 11110110 0	1fffffec [25]
(200)	11111111 11111111 11111000 10	3ffffe2 [26]
(201)	11111111 11111111 11111000 11	3ffffe3 [26]
(202)	11111111 11111111 11111001 00	3ffffe4 [26]
(203)	11111111 11111111 11111011 110	7ffffde [27]
(204)	11111111 11111111 11111011 111	7ffffdf [27]
(205)	11111111 11111111 11111001 01	3ffffe5 [26]
(206)	11111111 11111111 11110001	fffff1 [24]
(207)	11111111 11111111 11110110 1	1ffffed [25]
(208)	11111111 11111110 010	7fff2 [19]
(209)	11111111 11111111 00011	1ffffe3 [21]
(210)	11111111 11111111 11111001 10	3ffffe6 [26]
(211)	11111111 11111111 11111100 000	7ffffe0 [27]
(212)	11111111 11111111 11111100 001	7ffffe1 [27]
(213)	11111111 11111111 11111001 11	3ffffe7 [26]
(214)	11111111 11111111 11111100 010	7ffffe2 [27]
(215)	11111111 11111111 11110010	fffff2 [24]
(216)	11111111 11111111 00100	1ffffe4 [21]
(217)	11111111 11111111 00101	1ffffe5 [21]

(218)	11111111	11111111	11111010	00	3ffffe8	[26]
(219)	11111111	11111111	11111010	01	3ffffe9	[26]
(220)	11111111	11111111	11111111	1101	ffffffd	[28]
(221)	11111111	11111111	11111100	011	7ffffe3	[27]
(222)	11111111	11111111	11111100	100	7ffffe4	[27]
(223)	11111111	11111111	11111100	101	7ffffe5	[27]
(224)	11111111	11111111	1100		fffec	[20]
(225)	11111111	11111111	11110011		fffff3	[24]
(226)	11111111	11111111	1101		fffed	[20]
(227)	11111111	11111111	00110		1ffffe6	[21]
(228)	11111111	11111111	101001		3ffffe9	[22]
(229)	11111111	11111111	00111		1ffffe7	[21]
(230)	11111111	11111111	01000		1ffffe8	[21]
(231)	11111111	11111111	1110011		7fffff3	[23]
(232)	11111111	11111111	101010		3ffffea	[22]
(233)	11111111	11111111	101011		3ffffeb	[22]
(234)	11111111	11111111	11110111	0	1ffffee	[25]
(235)	11111111	11111111	11110111	1	1ffffef	[25]
(236)	11111111	11111111	11110100		fffff4	[24]
(237)	11111111	11111111	11110101		fffff5	[24]
(238)	11111111	11111111	11111010	10	3ffffea	[26]
(239)	11111111	11111111	1110100		7fffff4	[23]
(240)	11111111	11111111	11111010	11	3ffffeb	[26]
(241)	11111111	11111111	11111100	110	7ffffe6	[27]
(242)	11111111	11111111	11111011	00	3ffffec	[26]
(243)	11111111	11111111	11111011	01	3ffffed	[26]
(244)	11111111	11111111	11111100	111	7ffffe7	[27]
(245)	11111111	11111111	11111101	000	7ffffe8	[27]
(246)	11111111	11111111	11111101	001	7ffffe9	[27]
(247)	11111111	11111111	11111101	010	7ffffea	[27]
(248)	11111111	11111111	11111101	011	7ffffeb	[27]
(249)	11111111	11111111	11111111	1110	ffffffe	[28]
(250)	11111111	11111111	11111101	100	7ffffec	[27]
(251)	11111111	11111111	11111101	101	7ffffed	[27]
(252)	11111111	11111111	11111101	110	7ffffee	[27]
(253)	11111111	11111111	11111101	111	7ffffef	[27]
(254)	11111111	11111111	11111110	000	7fffff0	[27]
(255)	11111111	11111111	11111011	10	3ffffee	[26]
EOS (256)	11111111	11111111	11111111	111111	3fffffff	[30]

Appendix C. Examples

A number of examples are worked through here, covering integer encoding, header field representation, and the encoding of whole lists of header fields, for both requests and responses, and with and without Huffman coding.

C.1. Integer Representation Examples

This section shows the representation of integer values in details (see Section 5.1).

C.1.1. Example 1: Encoding 10 Using a 5-bit Prefix

The value 10 is to be encoded with a 5-bit prefix.

- o 10 is less than 31 ($2^5 - 1$) and is represented using the 5-bit prefix.

0	1	2	3	4	5	6	7	
X	X	X	0	1	0	1	0	10 stored on 5 bits

C.1.2. Example 2: Encoding 1337 Using a 5-bit Prefix

The value I=1337 is to be encoded with a 5-bit prefix.

1337 is greater than 31 ($2^5 - 1$).

The 5-bit prefix is filled with its max value (31).

$I = 1337 - (2^5 - 1) = 1306$.

I (1306) is greater than or equal to 128, the while loop body executes:

$I \% 128 == 26$

$26 + 128 == 154$

154 is encoded in 8 bits as: 10011010

I is set to 10 ($1306 / 128 == 10$)

I is no longer greater than or equal to 128, the while loop terminates.

I, now 10, is encoded in 8 bits as: 00001010.

The process ends.

0	1	2	3	4	5	6	7	
X	X	X	1	1	1	1	1	Prefix = 31, I = 1306
1	0	0	1	1	0	1	0	1306>=128, encode(154), I=1306/128
0	0	0	0	1	0	1	0	10<128, encode(10), done

C.1.3. Example 3: Encoding 42 Starting at an Octet Boundary

The value 42 is to be encoded starting at an octet-boundary. This implies that a 8-bit prefix is used.

- o 42 is less than 255 ($2^8 - 1$) and is represented using the 8-bit prefix.

0	1	2	3	4	5	6	7	
0	0	1	0	1	0	1	0	42 stored on 8 bits

C.2. Header Field Representation Examples

This section shows several independent representation examples.

C.2.1. Literal Header Field with Indexing

The header field representation uses a literal name and a literal value. The header field is added to the dynamic table.

Header list to encode:

custom-key: custom-header

Hex dump of encoded data:

400a 6375 7374 6f6d 2d6b 6579 0d63 7573	@.custom-key.cus
746f 6d2d 6865 6164 6572	tom-header

Decoding process:

40		== Literal indexed ==
0a		Literal name (len = 10)
6375 7374 6f6d 2d6b 6579		custom-key
0d		Literal value (len = 13)
6375 7374 6f6d 2d68 6561 6465 72		custom-header
		-> custom-key: custom-head\
		er

Dynamic Table (after decoding):

```
[ 1] (s = 55) custom-key: custom-header
      Table size: 55
```

Decoded header list:

custom-key: custom-header

C.2.2. Literal Header Field without Indexing

The header field representation uses an indexed name and a literal value. The header field is not added to the dynamic table.

Header list to encode:

:path: /sample/path

Hex dump of encoded data:

040c 2f73 616d 706c 652f 7061 7468		../sample/path
------------------------------------	--	----------------

Decoding process:

04		== Literal not indexed ==
		Indexed name (idx = 4)
		:path
0c		Literal value (len = 12)
2f73 616d 706c 652f 7061 7468		/sample/path
		-> :path: /sample/path

Dynamic table (after decoding): empty.

Decoded header list:

:path: /sample/path

C.2.3. Literal Header Field never Indexed

The header field representation uses a literal name and a literal value. The header field is not added to the dynamic table, and must use the same representation if re-encoded by an intermediary.

Header list to encode:

password: secret

Hex dump of encoded data:

```
1008 7061 7373 776f 7264 0673 6563 7265 | ..password.secre
74                                     | t
```

Decoding process:

```
10                                     | == Literal never indexed ==
08                                     |   Literal name (len = 8)
7061 7373 776f 7264                 | password
06                                     |   Literal value (len = 6)
7365 6372 6574                     | secret
                                     | -> password: secret
```

Dynamic table (after decoding): empty.

Decoded header list:

password: secret

C.2.4. Indexed Header Field

The header field representation uses an indexed header field, from the static table.

Header list to encode:

:method: GET

Hex dump of encoded data:

```
82                                     | .
```

Decoding process:

```
82                                     | == Indexed - Add ==
                                     |   idx = 2
                                     | -> :method: GET
```

Dynamic table (after decoding): empty.

Decoded header list:

:method: GET

C.3. Request Examples without Huffman Coding

This section shows several consecutive header lists, corresponding to HTTP requests, on the same connection.

C.3.1. First Request

Header list to encode:

:method: GET
:scheme: http
:path: /
:authority: www.example.com

Hex dump of encoded data:

8286 8441 0f77 7777 2e65 7861 6d70 6c65	...A.www.example
2e63 6f6d	.com

Decoding process:

82	== Indexed - Add ==
	idx = 2
	-> :method: GET
86	== Indexed - Add ==
	idx = 6
	-> :scheme: http
84	== Indexed - Add ==
	idx = 4
	-> :path: /
41	== Literal indexed ==
	Indexed name (idx = 1)
	:authority
0f	Literal value (len = 15)
7777 772e 6578 616d 706c 652e 636f 6d	www.example.com
	-> :authority: www.example\
	.com

Dynamic Table (after decoding):

[1] (s = 57) :authority: www.example.com
Table size: 57

Decoded header list:

```
:method: GET
:scheme: http
:path: /
:authority: www.example.com
```

C.3.2. Second Request

Header list to encode:

```
:method: GET
:scheme: http
:path: /
:authority: www.example.com
cache-control: no-cache
```

Hex dump of encoded data:

```
8286 84be 5808 6e6f 2d63 6163 6865      | ....X.no-cache
```

Decoding process:

82		== Indexed - Add ==
		idx = 2
		-> :method: GET
86		== Indexed - Add ==
		idx = 6
		-> :scheme: http
84		== Indexed - Add ==
		idx = 4
		-> :path: /
be		== Indexed - Add ==
		idx = 62
		-> :authority: www.example\
		.com
58		== Literal indexed ==
		Indexed name (idx = 24)
		cache-control
08		Literal value (len = 8)
6e6f 2d63 6163 6865		no-cache
		-> cache-control: no-cache

Dynamic Table (after decoding):

```
[ 1] (s = 53) cache-control: no-cache
[ 2] (s = 57) :authority: www.example.com
      Table size: 110
```

Decoded header list:

```
:method: GET
:scheme: http
:path: /
:authority: www.example.com
cache-control: no-cache
```

C.3.3. Third Request

Header list to encode:

```
:method: GET
:scheme: https
:path: /index.html
:authority: www.example.com
custom-key: custom-value
```

Hex dump of encoded data:

```
8287 85bf 400a 6375 7374 6f6d 2d6b 6579 | ....@.custom-key
0c63 7573 746f 6d2d 7661 6c75 65      | .custom-value
```

Decoding process:

82	== Indexed - Add ==
	idx = 2
	-> :method: GET
87	== Indexed - Add ==
	idx = 7
	-> :scheme: https
85	== Indexed - Add ==
	idx = 5
	-> :path: /index.html
bf	== Indexed - Add ==
	idx = 63
	-> :authority: www.example\
	.com
40	== Literal indexed ==
0a	Literal name (len = 10)
6375 7374 6f6d 2d6b 6579	custom-key
0c	Literal value (len = 12)
6375 7374 6f6d 2d76 616c 7565	custom-value
	-> custom-key: custom-valu\
	e

Dynamic Table (after decoding):

```
[ 1] (s = 54) custom-key: custom-value
[ 2] (s = 53) cache-control: no-cache
[ 3] (s = 57) :authority: www.example.com
      Table size: 164
```

Decoded header list:

```
:method: GET
:scheme: https
:path: /index.html
:authority: www.example.com
custom-key: custom-value
```

C.4. Request Examples with Huffman Coding

This section shows the same examples as the previous section, but using Huffman encoding for the literal values.

C.4.1. First Request

Header list to encode:

```
:method: GET
:scheme: http
:path: /
:authority: www.example.com
```

Hex dump of encoded data:

```
8286 8441 8cf1 e3c2 e5f2 3a6b a0ab 90f4 | ...A.....:k....
ff                                         | .
```


Decoding process:

82		== Indexed - Add ==
		idx = 2
		-> :method: GET
86		== Indexed - Add ==
		idx = 6
		-> :scheme: http
84		== Indexed - Add ==
		idx = 4
		-> :path: /
41		== Literal indexed ==
		Indexed name (idx = 1)
		:authority
8c		Literal value (len = 12)
		Huffman encoded:
f1e3 c2e5 f23a 6ba0 ab90 f4ff	:k.....
		Decoded:
		www.example.com
		-> :authority: www.example\
		.com

Dynamic Table (after decoding):

```
[ 1] (s = 57) :authority: www.example.com
      Table size: 57
```

Decoded header list:

```
:method: GET
:scheme: http
:path: /
:authority: www.example.com
```

C.4.2. Second Request

Header list to encode:

```
:method: GET
:scheme: http
:path: /
:authority: www.example.com
cache-control: no-cache
```

Hex dump of encoded data:

8286 84be 5886 a8eb 1064 9cbf	X....d..
-------------------------------	--	--------------

Decoding process:

82 86 84 be 58 86 a8eb 1064 9cbf	== Indexed - Add == idx = 2 -> :method: GET == Indexed - Add == idx = 6 -> :scheme: http == Indexed - Add == idx = 4 -> :path: / == Indexed - Add == idx = 62 -> :authority: www.example\ .com == Literal indexed == Indexed name (idx = 24) cache-control Literal value (len = 6) Huffman encoded: ...d.. Decoded: no-cache -> cache-control: no-cache
--	--

Dynamic Table (after decoding):

```
[ 1] (s = 53) cache-control: no-cache
[ 2] (s = 57) :authority: www.example.com
      Table size: 110
```

Decoded header list:

```
:method: GET
:scheme: http
:path: /
:authority: www.example.com
cache-control: no-cache
```

C.4.3. Third Request

Header list to encode:

```
:method: GET
:scheme: https
:path: /index.html
:authority: www.example.com
custom-key: custom-value
```

Hex dump of encoded data:

```
8287 85bf 4088 25a8 49e9 5ba9 7d7f 8925 | ....@.%.I.[.}..%
a849 e95b b8e8 b4bf | .I.[....
```

Decoding process:

```
82 | == Indexed - Add ==
   |     idx = 2
87 | -> :method: GET
   | == Indexed - Add ==
   |     idx = 7
   | -> :scheme: https
85 | == Indexed - Add ==
   |     idx = 5
   | -> :path: /index.html
bf | == Indexed - Add ==
   |     idx = 63
   | -> :authority: www.example\
   |     .com
40 | == Literal indexed ==
88 |     Literal name (len = 8)
   |     Huffman encoded:
25a8 49e9 5ba9 7d7f | %.I.[.}.
   |     Decoded:
   | custom-key
89 |     Literal value (len = 9)
   |     Huffman encoded:
25a8 49e9 5bb8 e8b4 bf | %.I.[....
   |     Decoded:
   | custom-value
   | -> custom-key: custom-valu\
   |     e
```

Dynamic Table (after decoding):

```
[ 1] (s = 54) custom-key: custom-value
[ 2] (s = 53) cache-control: no-cache
[ 3] (s = 57) :authority: www.example.com
    Table size: 164
```

Decoded header list:

```
:method: GET
:scheme: https
:path: /index.html
:authority: www.example.com
custom-key: custom-value
```

C.5. Response Examples without Huffman Coding

This section shows several consecutive header lists, corresponding to HTTP responses, on the same connection. The HTTP/2 setting parameter `SETTINGS_HEADER_TABLE_SIZE` is set to the value of 256 octets, causing some evictions to occur.

C.5.1. First Response

Header list to encode:

```
:status: 302
cache-control: private
date: Mon, 21 Oct 2013 20:13:21 GMT
location: https://www.example.com
```

Hex dump of encoded data:

4803	3330	3258	0770	7269	7661	7465	611d		H.302X.privatea.
4d6f	6e2c	2032	3120	4f63	7420	3230	3133		Mon, 21 Oct 2013
2032	303a	3133	3a32	3120	474d	546e	1768		20:13:21 GMTn.h
7474	7073	3a2f	2f77	7777	2e65	7861	6d70		ttps://www.examp
6c65	2e63	6f6d							le.com

Decoding process:

48 03 3330 32 58 07 7072 6976 6174 65 61 1d 4d6f 6e2c 2032 3120 4f63 7420 3230 3133 2032 303a 3133 3a32 3120 474d 54 6e 17 6874 7470 733a 2f2f 7777 772e 6578 616d 706c 652e 636f 6d	== Literal indexed == Indexed name (idx = 8) :status Literal value (len = 3) 302 -> :status: 302 == Literal indexed == Indexed name (idx = 24) cache-control Literal value (len = 7) private -> cache-control: private == Literal indexed == Indexed name (idx = 33) date Literal value (len = 29) Mon, 21 Oct 2013 20:13:21 GMT -> date: Mon, 21 Oct 2013 \ 20:13:21 GMT == Literal indexed == Indexed name (idx = 46) location Literal value (len = 23) https://www.exam ple.com -> location: https://www.e\ xample.com
---	--

Dynamic Table (after decoding):

```
[ 1] (s = 63) location: https://www.example.com
[ 2] (s = 65) date: Mon, 21 Oct 2013 20:13:21 GMT
[ 3] (s = 52) cache-control: private
[ 4] (s = 42) :status: 302
    Table size: 222
```

Decoded header list:

```
:status: 302
cache-control: private
date: Mon, 21 Oct 2013 20:13:21 GMT
location: https://www.example.com
```

C.5.2. Second Response

The (":status", "302") header field is evicted from the dynamic table to free space to allow adding the (":status", "307") header field.

Header list to encode:

```
:status: 307
cache-control: private
date: Mon, 21 Oct 2013 20:13:21 GMT
location: https://www.example.com
```

Hex dump of encoded data:

```
4803 3330 37c1 c0bf | H.307...
```

Decoding process:

48	== Literal indexed ==
	Indexed name (idx = 8)
	:status
03	Literal value (len = 3)
3330 37	307
	- evict: :status: 302
	-> :status: 307
c1	== Indexed - Add ==
	idx = 65
	-> cache-control: private
c0	== Indexed - Add ==
	idx = 64
	-> date: Mon, 21 Oct 2013 \
	20:13:21 GMT
bf	== Indexed - Add ==
	idx = 63
	-> location: https://www.e\
	xample.com

Dynamic Table (after decoding):

```
[ 1] (s = 42) :status: 307
[ 2] (s = 63) location: https://www.example.com
[ 3] (s = 65) date: Mon, 21 Oct 2013 20:13:21 GMT
[ 4] (s = 52) cache-control: private
    Table size: 222
```

Decoded header list:

```
:status: 307
cache-control: private
date: Mon, 21 Oct 2013 20:13:21 GMT
location: https://www.example.com
```

C.5.3. Third Response

Several header fields are evicted from the dynamic table during the processing of this header list.

Header list to encode:

```
:status: 200
cache-control: private
date: Mon, 21 Oct 2013 20:13:22 GMT
location: https://www.example.com
content-encoding: gzip
set-cookie: foo=ASDJKHQKBZXOQWEOPIUAXQWEOIU; max-age=3600; version=1
```

Hex dump of encoded data:

88c1	611d	4d6f	6e2c	2032	3120	4f63	7420		..a.Mon, 21 Oct
3230	3133	2032	303a	3133	3a32	3220	474d		2013 20:13:22 GM
54c0	5a04	677a	6970	7738	666f	6f3d	4153		T.Z.gzipw8foo=AS
444a	4b48	514b	425a	584f	5157	454f	5049		DJKHQKBZXOQWEOPI
5541	5851	5745	4f49	553b	206d	6178	2d61		UAXQWEOIU; max-a
6765	3d33	3630	303b	2076	6572	7369	6f6e		ge=3600; version
3d31									=1

Decoding process:

88	== Indexed - Add == idx = 8 -> :status: 200
c1	== Indexed - Add == idx = 65 -> cache-control: private
61	== Literal indexed == Indexed name (idx = 33) date
1d	Literal value (len = 29)
4d6f 6e2c 2032 3120 4f63 7420 3230 3133	Mon, 21 Oct 2013
2032 303a 3133 3a32 3220 474d 54	20:13:22 GMT
	- evict: cache-control: pr\ivate
	-> date: Mon, 21 Oct 2013 \20:13:22 GMT
c0	== Indexed - Add == idx = 64 -> location: https://www.e\ample.com
5a	== Literal indexed == Indexed name (idx = 26) content-encoding
04	Literal value (len = 4)
677a 6970	gzip
	- evict: date: Mon, 21 Oct\2013 20:13:21 GMT
	-> content-encoding: gzip
77	== Literal indexed == Indexed name (idx = 55) set-cookie
38	Literal value (len = 56)
666f 6f3d 4153 444a 4b48 514b 425a 584f	foo=ASDJKHQKBZXO
5157 454f 5049 5541 5851 5745 4f49 553b	QWEOPIUAXQWEOIU;
206d 6178 2d61 6765 3d33 3630 303b 2076	max-age=3600; v
6572 7369 6f6e 3d31	ersion=1
	- evict: location: https://\www.example.com
	- evict: :status: 307
	-> set-cookie: foo=ASDJKHQ\KBZXOQWEOPIUAXQWEOIU; ma\ x-age=3600; version=1

Dynamic Table (after decoding):

```
[ 1] (s = 98) set-cookie: foo=ASDJKHQKBZXOQWEOPIUAXQWEOIU; max-age\
    =3600; version=1
[ 2] (s = 52) content-encoding: gzip
[ 3] (s = 65) date: Mon, 21 Oct 2013 20:13:22 GMT
    Table size: 215
```

Decoded header list:

```
:status: 200
cache-control: private
date: Mon, 21 Oct 2013 20:13:22 GMT
location: https://www.example.com
content-encoding: gzip
set-cookie: foo=ASDJKHQKBZXOQWEOPIUAXQWEOIU; max-age=3600; version=1
```

C.6. Response Examples with Huffman Coding

This section shows the same examples as the previous section, but using Huffman encoding for the literal values. The HTTP/2 setting parameter `SETTINGS_HEADER_TABLE_SIZE` is set to the value of 256 octets, causing some evictions to occur. The eviction mechanism uses the length of the decoded literal values, so the same evictions occurs as in the previous section.

C.6.1. First Response

Header list to encode:

```
:status: 302
cache-control: private
date: Mon, 21 Oct 2013 20:13:21 GMT
location: https://www.example.com
```

Hex dump of encoded data:

4882	6402	5885	aec3	771a	4b61	96d0	7abe		H.d.X...w.Ka..z.
9410	54d4	44a8	2005	9504	0b81	66e0	82a6		..T.D.f...
2dlb	ff6e	919d	29ad	1718	63c7	8f0b	97c8		-..n..)....c.....
e9ae	82ae	43d3						C.

Decoding process:

48	82	6402	58	85	aec3 771a 4b	61	96	d07a be94 1054 d444 a820 0595 040b 8166 e082 a62d 1bff	6e	91	9d29 ad17 1863 c78f 0b97 c8e9 ae82 ae43 d3	<pre> == Literal indexed == Indexed name (idx = 8) :status Literal value (len = 2) Huffman encoded: d. Decoded: 302 -> :status: 302 == Literal indexed == Indexed name (idx = 24) cache-control Literal value (len = 5) Huffman encoded: ..w.K Decoded: private -> cache-control: private == Literal indexed == Indexed name (idx = 33) date Literal value (len = 22) Huffman encoded: .z...T.D.f ...-... Decoded: Mon, 21 Oct 2013 20:13:21 \ GMT -> date: Mon, 21 Oct 2013 \ 20:13:21 GMT == Literal indexed == Indexed name (idx = 46) location Literal value (len = 17) Huffman encoded: .)...c.....C . Decoded: https://www.example.com -> location: https://www.e\ xample.com </pre>
----	----	------	----	----	--------------	----	----	---	----	----	---	---

Dynamic Table (after decoding):

```
[ 1] (s = 63) location: https://www.example.com
[ 2] (s = 65) date: Mon, 21 Oct 2013 20:13:21 GMT
[ 3] (s = 52) cache-control: private
[ 4] (s = 42) :status: 302
    Table size: 222
```

Decoded header list:

```
:status: 302
cache-control: private
date: Mon, 21 Oct 2013 20:13:21 GMT
location: https://www.example.com
```

C.6.2. Second Response

The (":status", "302") header field is evicted from the dynamic table to free space to allow adding the (":status", "307") header field.

Header list to encode:

```
:status: 307
cache-control: private
date: Mon, 21 Oct 2013 20:13:21 GMT
location: https://www.example.com
```

Hex dump of encoded data:

```
4883 640e ffc1 c0bf          | H.d.....
```

Decoding process:

48		== Literal indexed ==
		Indexed name (idx = 8)
		:status
83		Literal value (len = 3)
		Huffman encoded:
640e ff		d..
		Decoded:
		307
		- evict: :status: 302
		-> :status: 307
c1		== Indexed - Add ==
		idx = 65
		-> cache-control: private
c0		== Indexed - Add ==
		idx = 64
		-> date: Mon, 21 Oct 2013 \
		20:13:21 GMT
bf		== Indexed - Add ==
		idx = 63
		-> location: https://www.e\
		xample.com

Dynamic Table (after decoding):

```
[ 1] (s = 42) :status: 307
[ 2] (s = 63) location: https://www.example.com
[ 3] (s = 65) date: Mon, 21 Oct 2013 20:13:21 GMT
[ 4] (s = 52) cache-control: private
      Table size: 222
```

Decoded header list:

```
:status: 307
cache-control: private
date: Mon, 21 Oct 2013 20:13:21 GMT
location: https://www.example.com
```

C.6.3. Third Response

Several header fields are evicted from the dynamic table during the processing of this header list.

Header list to encode:

```
:status: 200
cache-control: private
date: Mon, 21 Oct 2013 20:13:22 GMT
location: https://www.example.com
content-encoding: gzip
set-cookie: foo=ASDJKHQBZXOQWEOPIUAXQWEOIU; max-age=3600; version=1
```

Hex dump of encoded data:

88c1 6196 d07a be94 1054 d444 a820 0595	..a...z...T.D. ...
040b 8166 e084 a62d 1bff c05a 839b d9ab	...f...-...Z....
77ad 94e7 821d d7f2 e6c7 b335 dfdf cd5b	w.....5...[
3960 d5af 2708 7f36 72c1 ab27 0fb5 291f	9\'...\'..6r...\'..).
9587 3160 65c0 03ed 4ee5 b106 3d50 07	..1\'e...N...=P.

Decoding process:

88	== Indexed - Add ==
	idx = 8
	-> :status: 200
c1	== Indexed - Add ==
	idx = 65
	-> cache-control: private
61	== Literal indexed ==
	Indexed name (idx = 33)
	date
96	Literal value (len = 22)
	Huffman encoded:
d07a be94 1054 d444 a820 0595 040b 8166	.z...T.D.f
e084 a62d 1bff	...-...
	Decoded:
	Mon, 21 Oct 2013 20:13:22 \
	GMT
	- evict: cache-control: pr\
	ivate
	-> date: Mon, 21 Oct 2013 \
	20:13:22 GMT
c0	== Indexed - Add ==
	idx = 64
	-> location: https://www.e\
	xample.com
5a	== Literal indexed ==
	Indexed name (idx = 26)
	content-encoding
83	Literal value (len = 3)
	Huffman encoded:

<pre> 9bd9 ab 77 ad 94e7 821d d7f2 e6c7 b335 dfdf cd5b 3960 d5af 2708 7f36 72c1 ab27 0fb5 291f 9587 3160 65c0 03ed 4ee5 b106 3d50 07 </pre>	<pre> ... Decoded: gzip - evict: date: Mon, 21 Oct\ 2013 20:13:21 GMT -> content-encoding: gzip == Literal indexed == Indexed name (idx = 55) set-cookie Literal value (len = 45) Huffman encoded: 5...[9` ..'...6r..'')... 1'e...N...=P. Decoded: foo=ASDJKHQKBZXOQWEOPIUAXQ\ WEOIU; max-age=3600; versi\ on=1 - evict: location: https://\ /www.example.com - evict: :status: 307 -> set-cookie: foo=ASDJKHQ\ KBZXOQWEOPIUAXQWEOIU; ma\ x-age=3600; version=1 </pre>
---	---

Dynamic Table (after decoding):

```

[ 1] (s = 98) set-cookie: foo=ASDJKHQKBZXOQWEOPIUAXQWEOIU; max-age\
      =3600; version=1
[ 2] (s = 52) content-encoding: gzip
[ 3] (s = 65) date: Mon, 21 Oct 2013 20:13:22 GMT
      Table size: 215

```

Decoded header list:

```

:status: 200
cache-control: private
date: Mon, 21 Oct 2013 20:13:22 GMT
location: https://www.example.com
content-encoding: gzip
set-cookie: foo=ASDJKHQKBZXOQWEOPIUAXQWEOIU; max-age=3600; version=1

```

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

- D.1. Since draft-ietf-httpbis-header-compression-10
- o Editorial corrections for taking into account IETF LC comments.
 - * Added links to security sections.
 - * Made spec more independent of HTTP/2.
 - * Expanded security section about never indexed literal usage.
 - o Removed most usages of 'name-value pair' instead of header field.
 - o Changed 'header table' to 'header field table'.
- D.2. Since draft-ietf-httpbis-header-compression-09
- o Renamed header table to dynamic table.
 - o Updated integer representation.
 - o Editorial corrections.
- D.3. Since draft-ietf-httpbis-header-compression-08
- o Removed the reference set.
 - o Removed header emission.
 - o Explicit handling of several SETTINGS_HEADER_TABLE_SIZE parameter changes.
 - o Changed header set to header list, and forced ordering.
 - o Updated examples.
 - o Exchanged header and static table positions.
- D.4. Since draft-ietf-httpbis-header-compression-07
- o Removed old text on index value of 0.
 - o Added clarification for signalling of maximum table size after a SETTINGS_HEADER_TABLE_SIZE update.
 - o Rewrote security considerations.
 - o Many editorial clarifications or improvements.

- o Added convention section.
- o Reworked document's outline.
- o Updated static table. Entry 16 has now "gzip, deflate" for value.
- o Updated Huffman table, using data set provided by Google.

D.5. Since draft-ietf-httpbis-header-compression-06

- o Updated format to include literal headers that must never be compressed.
- o Updated security considerations.
- o Moved integer encoding examples to the appendix.
- o Updated Huffman table.
- o Updated static header table (adding and removing status values).
- o Updated examples.

D.6. Since draft-ietf-httpbis-header-compression-05

- o Regenerated examples.
- o Only one Huffman table for requests and responses.
- o Added maximum size for dynamic table, independent of SETTINGS_HEADER_TABLE_SIZE.
- o Added pseudo-code for integer decoding.
- o Improved examples (removing unnecessary removals).

D.7. Since draft-ietf-httpbis-header-compression-04

- o Updated examples: take into account changes in the spec, and show more features.
- o Use 'octet' everywhere instead of having both 'byte' and 'octet'.
- o Added reference set emptying.
- o Editorial changes and clarifications.
- o Added "host" header to the static table.

- o Ordering for list of values (either NULL- or comma-separated).
- D.8. Since draft-ietf-httpbis-header-compression-03
- o A large number of editorial changes; changed the description of evicting/adding new entries.
 - o Removed substitution indexing
 - o Changed 'initial headers' to 'static headers', as per issue #258
 - o Merged 'request' and 'response' static headers, as per issue #259
 - o Changed text to indicate that new headers are added at index 0 and expire from the largest index, as per issue #233
- D.9. Since draft-ietf-httpbis-header-compression-02
- o Corrected error in integer encoding pseudocode.
- D.10. Since draft-ietf-httpbis-header-compression-01
- o Refactored of Header Encoding Section: split definitions and processing rule.
 - o Backward incompatible change: Updated reference set management as per issue #214. This changes how the interaction between the reference set and eviction works. This also changes the working of the reference set in some specific cases.
 - o Backward incompatible change: modified initial header list, as per issue #188.
 - o Added example of 32 octets entry structure (issue #191).
 - o Added Header Set Completion section. Reflowed some text. Clarified some writing which was awkward. Added text about duplicate header entry encoding. Clarified some language w.r.t Header Set. Changed x-my-header to mynewheader. Added text in the HeaderEmission section indicating that the application may also be able to free up memory more quickly. Added information in Security Considerations section.
- D.11. Since draft-ietf-httpbis-header-compression-00
- Fixed bug/omission in integer representation algorithm.
- Changed the document title.

Header matching text rewritten.

Changed the definition of header emission.

Changed the name of the setting which dictates how much memory the compression context should use.

Removed "specific use cases" section

Corrected erroneous statement about what index can be contained in one octet

Added descriptions of opcodes

Removed security claims from introduction.

Authors' Addresses

Roberto Peon
Google, Inc

EMail: fenix@google.com

Herve Ruellan
Canon CRF

EMail: herve.ruellan@crf.canon.fr

HTTPbis Working Group
Internet-Draft
Intended status: Standards Track
Expires: August 15, 2015

M. Belshe
Twist
R. Peon
Google, Inc
M. Thomson, Ed.
Mozilla
February 11, 2015

Hypertext Transfer Protocol version 2
draft-ietf-httpbis-http2-17

Abstract

This specification describes an optimized expression of the semantics of the Hypertext Transfer Protocol (HTTP). HTTP/2 enables a more efficient use of network resources and a reduced perception of latency by introducing header field compression and allowing multiple concurrent exchanges on the same connection. It also introduces unsolicited push of representations from servers to clients.

This specification is an alternative to, but does not obsolete, the HTTP/1.1 message syntax. HTTP's existing semantics remain unchanged.

Editorial Note (To be removed by RFC Editor)

Discussion of this draft takes place on the HTTPBIS working group mailing list (ietf-http-wg@w3.org), which is archived at [1].

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

The changes in this draft are summarized in Appendix B.

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 August 15, 2015.

Copyright Notice

Copyright (c) 2015 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction	4
2. HTTP/2 Protocol Overview	5
2.1. Document Organization	6
2.2. Conventions and Terminology	6
3. Starting HTTP/2	7
3.1. HTTP/2 Version Identification	8
3.2. Starting HTTP/2 for "http" URIs	9
3.2.1. HTTP2-Settings Header Field	10
3.3. Starting HTTP/2 for "https" URIs	11
3.4. Starting HTTP/2 with Prior Knowledge	11
3.5. HTTP/2 Connection Preface	11
4. HTTP Frames	12
4.1. Frame Format	13
4.2. Frame Size	14
4.3. Header Compression and Decompression	14
5. Streams and Multiplexing	15
5.1. Stream States	16
5.1.1. Stream Identifiers	21
5.1.2. Stream Concurrency	22
5.2. Flow Control	23
5.2.1. Flow Control Principles	23
5.2.2. Appropriate Use of Flow Control	24
5.3. Stream priority	25
5.3.1. Stream Dependencies	25
5.3.2. Dependency Weighting	26
5.3.3. Reprioritization	27
5.3.4. Prioritization State Management	27
5.3.5. Default Priorities	29
5.4. Error Handling	29

5.4.1.	Connection Error Handling	29
5.4.2.	Stream Error Handling	30
5.4.3.	Connection Termination	30
5.5.	Extending HTTP/2	30
6.	Frame Definitions	31
6.1.	DATA	31
6.2.	HEADERS	33
6.3.	PRIORITY	35
6.4.	RST_STREAM	36
6.5.	SETTINGS	37
6.5.1.	SETTINGS Format	38
6.5.2.	Defined SETTINGS Parameters	38
6.5.3.	Settings Synchronization	40
6.6.	PUSH_PROMISE	40
6.7.	PING	42
6.8.	GOAWAY	43
6.9.	WINDOW_UPDATE	46
6.9.1.	The Flow Control Window	47
6.9.2.	Initial Flow Control Window Size	48
6.9.3.	Reducing the Stream Window Size	49
6.10.	CONTINUATION	49
7.	Error Codes	50
8.	HTTP Message Exchanges	51
8.1.	HTTP Request/Response Exchange	51
8.1.1.	Upgrading From HTTP/2	53
8.1.2.	HTTP Header Fields	53
8.1.3.	Examples	57
8.1.4.	Request Reliability Mechanisms in HTTP/2	59
8.2.	Server Push	60
8.2.1.	Push Requests	61
8.2.2.	Push Responses	62
8.3.	The CONNECT Method	63
9.	Additional HTTP Requirements/Considerations	64
9.1.	Connection Management	64
9.1.1.	Connection Reuse	65
9.1.2.	The 421 (Misdirected Request) Status Code	66
9.2.	Use of TLS Features	66
9.2.1.	TLS 1.2 Features	67
9.2.2.	TLS 1.2 Cipher Suites	68
10.	Security Considerations	68
10.1.	Server Authority	68
10.2.	Cross-Protocol Attacks	68
10.3.	Intermediary Encapsulation Attacks	69
10.4.	Cacheability of Pushed Responses	69
10.5.	Denial of Service Considerations	70
10.5.1.	Limits on Header Block Size	71
10.5.2.	CONNECT Issues	71
10.6.	Use of Compression	72

10.7.	Use of Padding	72
10.8.	Privacy Considerations	73
11.	IANA Considerations	73
11.1.	Registration of HTTP/2 Identification Strings	74
11.2.	Frame Type Registry	74
11.3.	Settings Registry	75
11.4.	Error Code Registry	76
11.5.	HTTP2-Settings Header Field Registration	77
11.6.	PRI Method Registration	78
11.7.	The 421 (Misdirected Request) HTTP Status Code	78
12.	Acknowledgements	78
13.	References	79
13.1.	Normative References	79
13.2.	Informative References	80
13.3.	URIs	81
Appendix A.	TLS 1.2 Cipher Suite Black List	82
Appendix B.	Change Log	86
B.1.	Since draft-ietf-httpbis-http2-15	86
B.2.	Since draft-ietf-httpbis-http2-14	86
B.3.	Since draft-ietf-httpbis-http2-13	87
B.4.	Since draft-ietf-httpbis-http2-12	87
B.5.	Since draft-ietf-httpbis-http2-11	87
B.6.	Since draft-ietf-httpbis-http2-10	87
B.7.	Since draft-ietf-httpbis-http2-09	88
B.8.	Since draft-ietf-httpbis-http2-08	88
B.9.	Since draft-ietf-httpbis-http2-07	89
B.10.	Since draft-ietf-httpbis-http2-06	89
B.11.	Since draft-ietf-httpbis-http2-05	89
B.12.	Since draft-ietf-httpbis-http2-04	89
B.13.	Since draft-ietf-httpbis-http2-03	90
B.14.	Since draft-ietf-httpbis-http2-02	90
B.15.	Since draft-ietf-httpbis-http2-01	90
B.16.	Since draft-ietf-httpbis-http2-00	91
B.17.	Since draft-mbelshe-httpbis-spdyl-00	91

1. Introduction

The Hypertext Transfer Protocol (HTTP) is a wildly successful protocol. However, how HTTP/1.1 uses the underlying transport ([RFC7230], Section 6) has several characteristics that have a negative overall effect on application performance today.

In particular, HTTP/1.0 allowed only one request to be outstanding at a time on a given TCP connection. HTTP/1.1 added request pipelining, but this only partially addressed request concurrency and still suffers from head-of-line blocking. Therefore, HTTP/1.0 and HTTP/1.1 clients that need to make many requests use multiple connections to a server in order to achieve concurrency and thereby reduce latency.

Furthermore, HTTP header fields are often repetitive and verbose, causing unnecessary network traffic, as well as causing the initial TCP [TCP] congestion window to quickly fill. This can result in excessive latency when multiple requests are made on a new TCP connection.

HTTP/2 addresses these issues by defining an optimized mapping of HTTP's semantics to an underlying connection. Specifically, it allows interleaving of request and response messages on the same connection and uses an efficient coding for HTTP header fields. It also allows prioritization of requests, letting more important requests complete more quickly, further improving performance.

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

Finally, HTTP/2 also enables more efficient processing of messages through use of binary message framing.

2. HTTP/2 Protocol Overview

HTTP/2 provides an optimized transport for HTTP semantics. HTTP/2 supports all of the core features of HTTP/1.1, but aims to be more efficient in several ways.

The basic protocol unit in HTTP/2 is a frame (Section 4.1). Each frame type serves a different purpose. For example, HEADERS and DATA frames form the basis of HTTP requests and responses (Section 8.1); other frame types like SETTINGS, WINDOW_UPDATE, and PUSH_PROMISE are used in support of other HTTP/2 features.

Multiplexing of requests is achieved by having each HTTP request-response exchange associated with its own stream (Section 5). Streams are largely independent of each other, so a blocked or stalled request or response does not prevent progress on other streams.

Flow control and prioritization ensure that it is possible to efficiently use multiplexed streams. Flow control (Section 5.2) helps to ensure that only data that can be used by a receiver is transmitted. Prioritization (Section 5.3) ensures that limited resources can be directed to the most important streams first.

HTTP/2 adds a new interaction mode, whereby a server can push responses to a client (Section 8.2). Server push allows a server to

speculatively send data to a client that the server anticipates the client will need, trading off some network usage against a potential latency gain. The server does this by synthesizing a request, which it sends as a PUSH_PROMISE frame. The server is then able to send a response to the synthetic request on a separate stream.

Because HTTP header fields used in a connection can contain large amounts of redundant data, frames that contain them are compressed (Section 4.3). This has especially advantageous impact upon request sizes in the common case, allowing many requests to be compressed into one packet.

2.1. Document Organization

The HTTP/2 specification is split into four parts:

- o Starting HTTP/2 (Section 3) covers how an HTTP/2 connection is initiated.
- o The framing (Section 4) and streams (Section 5) layers describe the way HTTP/2 frames are structured and formed into multiplexed streams.
- o Frame (Section 6) and error (Section 7) definitions include details of the frame and error types used in HTTP/2.
- o HTTP mappings (Section 8) and additional requirements (Section 9) describe how HTTP semantics are expressed using frames and streams.

While some of the frame and stream layer concepts are isolated from HTTP, this specification does not define a completely generic framing layer. The framing and streams layers are tailored to the needs of the HTTP protocol and server push.

2.2. Conventions and Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

All numeric values are in network byte order. Values are unsigned unless otherwise indicated. Literal values are provided in decimal or hexadecimal as appropriate. Hexadecimal literals are prefixed with "0x" to distinguish them from decimal literals.

The following terms are used:

client: The endpoint that initiates an HTTP/2 connection. Clients send HTTP requests and receive HTTP responses.

connection: A transport-layer connection between two endpoints.

connection error: An error that affects the entire HTTP/2 connection.

endpoint: Either the client or server of the connection.

frame: The smallest unit of communication within an HTTP/2 connection, consisting of a header and a variable-length sequence of octets structured according to the frame type.

peer: An endpoint. When discussing a particular endpoint, "peer" refers to the endpoint that is remote to the primary subject of discussion.

receiver: An endpoint that is receiving frames.

sender: An endpoint that is transmitting frames.

server: The endpoint that accepts an HTTP/2 connection. Servers receive HTTP requests and serve HTTP responses.

stream: A bi-directional flow of frames within the HTTP/2 connection.

stream error: An error on the individual HTTP/2 stream.

Finally, the terms "gateway", "intermediary", "proxy", and "tunnel" are defined in Section 2.3 of [RFC7230]. Intermediaries act as both client and server at different times.

The term "payload body" is defined in Section 3.3 of [RFC7230].

3. Starting HTTP/2

An HTTP/2 connection is an application layer protocol running on top of a TCP connection ([TCP]). The client is the TCP connection initiator.

HTTP/2 uses the same "http" and "https" URI schemes used by HTTP/1.1. HTTP/2 shares the same default port numbers: 80 for "http" URIs and 443 for "https" URIs. As a result, implementations processing requests for target resource URIs like "http://example.org/foo" or "https://example.com/bar" are required to first discover whether the

upstream server (the immediate peer to which the client wishes to establish a connection) supports HTTP/2.

The means by which support for HTTP/2 is determined is different for "http" and "https" URIs. Discovery for "http" URIs is described in Section 3.2. Discovery for "https" URIs is described in Section 3.3.

3.1. HTTP/2 Version Identification

The protocol defined in this document has two identifiers.

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

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

- o The string "h2c" identifies the protocol where HTTP/2 is run over cleartext TCP. This identifier is used in the HTTP/1.1 Upgrade header field and in any place where HTTP/2 over TCP is identified.

The "h2c" string is reserved from the ALPN identifier space, but describes a protocol that does not use TLS.

Negotiating "h2" or "h2c" implies the use of the transport, security, framing and message semantics described in this document.

[[CREF1: RFC Editor's Note: please remove the remainder of this section prior to the publication of a final version of this document.]]

Only implementations of the final, published RFC can identify themselves as "h2" or "h2c". Until such an RFC exists, implementations MUST NOT identify themselves using these strings.

Implementations of draft versions of the protocol MUST add the string "-" and the corresponding draft number to the identifier. For example, draft-ietf-httpbis-http2-11 over TLS is identified using the string "h2-11".

Non-compatible experiments that are based on these draft versions MUST append the string "-" and an experiment name to the identifier. For example, an experimental implementation of packet mood-based encoding based on draft-ietf-httpbis-http2-09 might identify itself as "h2-09-emo". Note that any label MUST conform to the "token" syntax defined in Section 3.2.6 of [RFC7230]. Experimenters are

encouraged to coordinate their experiments on the ietf-http-wg@w3.org mailing list.

3.2. Starting HTTP/2 for "http" URIs

A client that makes a request for an "http" URI without prior knowledge about support for HTTP/2 on the next hop uses the HTTP Upgrade mechanism (Section 6.7 of [RFC7230]). The client does so by making an HTTP/1.1 request that includes an Upgrade header field with the "h2c" token. Such an HTTP/1.1 request MUST include exactly one HTTP2-Settings (Section 3.2.1) header field.

For example:

```
GET / HTTP/1.1
Host: server.example.com
Connection: Upgrade, HTTP2-Settings
Upgrade: h2c
HTTP2-Settings: <base64url encoding of HTTP/2 SETTINGS payload>
```

Requests that contain a payload body MUST be sent in their entirety before the client can send HTTP/2 frames. This means that a large request can block the use of the connection until it is completely sent.

If concurrency of an initial request with subsequent requests is important, an OPTIONS request can be used to perform the upgrade to HTTP/2, at the cost of an additional round-trip.

A server that does not support HTTP/2 can respond to the request as though the Upgrade header field were absent:

```
HTTP/1.1 200 OK
Content-Length: 243
Content-Type: text/html
```

...

A server MUST ignore an "h2" token in an Upgrade header field. Presence of a token with "h2" implies HTTP/2 over TLS, which is instead negotiated as described in Section 3.3.

A server that supports HTTP/2 accepts the upgrade with a 101 (Switching Protocols) response. After the empty line that terminates the 101 response, the server can begin sending HTTP/2 frames. These frames MUST include a response to the request that initiated the Upgrade.

For example:

```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: h2c
```

```
[ HTTP/2 connection ...
```

The first HTTP/2 frame sent by the server MUST be a SETTINGS frame (Section 6.5) as the server connection preface (Section 3.5). Upon receiving the 101 response, the client MUST send a connection preface (Section 3.5), which includes a SETTINGS frame.

The HTTP/1.1 request that is sent prior to upgrade is assigned a stream identifier of 1 (see Section 5.1.1) with default priority values (Section 5.3.5). Stream 1 is implicitly "half closed" from the client toward the server (see Section 5.1), since the request is completed as an HTTP/1.1 request. After commencing the HTTP/2 connection, stream 1 is used for the response.

3.2.1. HTTP2-Settings Header Field

A request that upgrades from HTTP/1.1 to HTTP/2 MUST include exactly one "HTTP2-Settings" header field. The "HTTP2-Settings" header field is a connection-specific header field that includes parameters that govern the HTTP/2 connection, provided in anticipation of the server accepting the request to upgrade.

```
HTTP2-Settings    = token68
```

A server MUST NOT upgrade the connection to HTTP/2 if this header field is not present, or if more than one is present. A server MUST NOT send this header field.

The content of the "HTTP2-Settings" header field is the payload of a SETTINGS frame (Section 6.5), encoded as a base64url string (that is, the URL- and filename-safe Base64 encoding described in Section 5 of [RFC4648], with any trailing '=' characters omitted). The ABNF [RFC5234] production for "token68" is defined in Section 2.1 of [RFC7235].

Since the upgrade is only intended to apply to the immediate connection, a client sending "HTTP2-Settings" MUST also send "HTTP2-Settings" as a connection option in the "Connection" header field to prevent it from being forwarded (see Section 6.1 of [RFC7230]).

A server decodes and interprets these values as it would any other SETTINGS frame. Explicit acknowledgement of these settings (Section 6.5.3) is not necessary, since a 101 response serves as implicit acknowledgment. Providing these values in the Upgrade request gives a client an opportunity to provide parameters prior to receiving any frames from the server.

3.3. Starting HTTP/2 for "https" URIs

A client that makes a request to an "https" URI uses TLS [TLS12] with the application layer protocol negotiation (ALPN) extension [TLS-ALPN].

HTTP/2 over TLS uses the "h2" protocol identifier. The "h2c" protocol identifier MUST NOT be sent by a client or selected by a server; the "h2c" protocol identifier describes a protocol that does not use TLS.

Once TLS negotiation is complete, both the client and the server MUST send a connection preface (Section 3.5).

3.4. Starting HTTP/2 with Prior Knowledge

A client can learn that a particular server supports HTTP/2 by other means. For example, [ALT-SVC] describes a mechanism for advertising this capability.

A client MUST send the connection preface (Section 3.5), and then MAY immediately send HTTP/2 frames to such a server; servers can identify these connections by the presence of the connection preface. This only affects the establishment of HTTP/2 connections over cleartext TCP; implementations that support HTTP/2 over TLS MUST use protocol negotiation in TLS [TLS-ALPN].

Likewise, the server MUST send a connection preface (Section 3.5).

Without additional information, prior support for HTTP/2 is not a strong signal that a given server will support HTTP/2 for future connections. For example, it is possible for server configurations to change, for configurations to differ between instances in clustered servers, or for network conditions to change.

3.5. HTTP/2 Connection Preface

In HTTP/2, each endpoint is required to send a connection preface as a final confirmation of the protocol in use, and to establish the initial settings for the HTTP/2 connection. The client and server each send a different connection preface.

The client connection preface starts with a sequence of 24 octets, which in hex notation are:

```
0x505249202a20485454502f322e300d0a0d0a534d0d0a0d0a
```

(the string "PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n"). This sequence MUST be followed by a SETTINGS frame (Section 6.5), which MAY be empty. The client sends the client connection preface immediately upon receipt of a 101 Switching Protocols response (indicating a successful upgrade), or as the first application data octets of a TLS connection. If starting an HTTP/2 connection with prior knowledge of server support for the protocol, the client connection preface is sent upon connection establishment.

The client connection preface is selected so that a large proportion of HTTP/1.1 or HTTP/1.0 servers and intermediaries do not attempt to process further frames. Note that this does not address the concerns raised in [TALKING].

The server connection preface consists of a potentially empty SETTINGS frame (Section 6.5) that MUST be the first frame the server sends in the HTTP/2 connection.

The SETTINGS frames received from a peer as part of the connection preface MUST be acknowledged (see Section 6.5.3) after sending the connection preface.

To avoid unnecessary latency, clients are permitted to send additional frames to the server immediately after sending the client connection preface, without waiting to receive the server connection preface. It is important to note, however, that the server connection preface SETTINGS frame might include parameters that necessarily alter how a client is expected to communicate with the server. Upon receiving the SETTINGS frame, the client is expected to honor any parameters established. In some configurations, it is possible for the server to transmit SETTINGS before the client sends additional frames, providing an opportunity to avoid this issue.

Clients and servers MUST treat an invalid connection preface as a connection error (Section 5.4.1) of type `PROTOCOL_ERROR`. A GOAWAY frame (Section 6.8) MAY be omitted in this case, since an invalid preface indicates that the peer is not using HTTP/2.

4. HTTP Frames

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

4.1. Frame Format

All frames begin with a fixed 9-octet header followed by a variable-length payload.

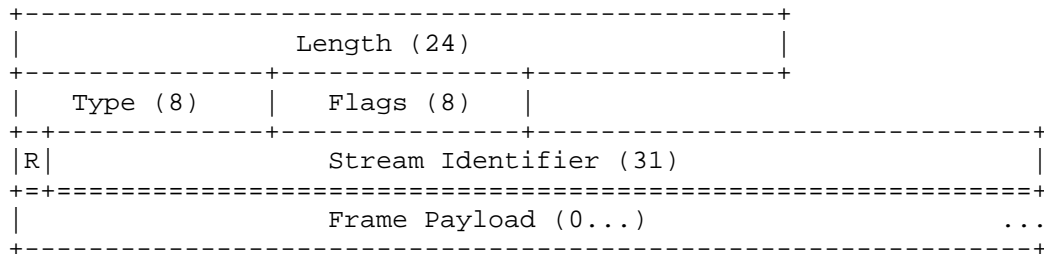


Figure 1: Frame Layout

The fields of the frame header are defined as:

Length: The length of the frame payload expressed as an unsigned 24-bit integer. Values greater than 2^{14} (16,384) MUST NOT be sent unless the receiver has set a larger value for `SETTINGS_MAX_FRAME_SIZE`.

The 9 octets of the frame header are not included in this value.

Type: The 8-bit type of the frame. The frame type determines the format and semantics of the frame. Implementations MUST ignore and discard any frame that has a type that is unknown.

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

Flags are assigned semantics specific to the indicated frame type. Flags that have no defined semantics for a particular frame type MUST be ignored, and MUST be left unset (0x0) when sending.

R: A reserved 1-bit field. The semantics of this bit are undefined and the bit MUST remain unset (0x0) when sending and MUST be ignored when receiving.

Stream Identifier: A stream identifier (see Section 5.1.1) expressed as an unsigned 31-bit integer. The value 0x0 is reserved for frames that are associated with the connection as a whole as opposed to an individual stream.

The structure and content of the frame payload is dependent entirely on the frame type.

4.2. Frame Size

The size of a frame payload is limited by the maximum size that a receiver advertises in the `SETTINGS_MAX_FRAME_SIZE` setting. This setting can have any value between 2^{14} (16,384) and $2^{24}-1$ (16,777,215) octets, inclusive.

All implementations **MUST** be capable of receiving and minimally processing frames up to 2^{14} octets in length, plus the 9 octet frame header (Section 4.1). The size of the frame header is not included when describing frame sizes.

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

An endpoint **MUST** send a `FRAME_SIZE_ERROR` error if a frame exceeds the size defined in `SETTINGS_MAX_FRAME_SIZE`, any limit defined for the frame type, or it is too small to contain mandatory frame data. A frame size error in a frame that could alter the state of the entire connection **MUST** be treated as a connection error (Section 5.4.1); this includes any frame carrying a header block (Section 4.3) (that is, `HEADERS`, `PUSH_PROMISE`, and `CONTINUATION`), `SETTINGS`, and any frame with a stream identifier of 0.

Endpoints are not obligated to use all available space in a frame. Responsiveness can be improved by using frames that are smaller than the permitted maximum size. Sending large frames can result in delays in sending time-sensitive frames (such as `RST_STREAM`, `WINDOW_UPDATE`, or `PRIORITY`) which if blocked by the transmission of a large frame, could affect performance.

4.3. Header Compression and Decompression

Just as in HTTP/1, a header field in HTTP/2 is a name with one or more associated values. They are used within HTTP request and response messages as well as server push operations (see Section 8.2).

Header lists are collections of zero or more header fields. When transmitted over a connection, a header list is serialized into a header block using HTTP Header Compression [`COMPRESSION`]. The serialized header block is then divided into one or more octet sequences, called header block fragments, and transmitted within the payload of `HEADERS` (Section 6.2), `PUSH_PROMISE` (Section 6.6) or `CONTINUATION` (Section 6.10) frames.

The Cookie header field [`COOKIE`] is treated specially by the HTTP mapping (see Section 8.1.2.5).

A receiving endpoint reassembles the header block by concatenating its fragments, then decompresses the block to reconstruct the header list.

A complete header block consists of either:

- o a single HEADERS or PUSH_PROMISE frame, with the END_HEADERS flag set, or
- o a HEADERS or PUSH_PROMISE frame with the END_HEADERS flag cleared and one or more CONTINUATION frames, where the last CONTINUATION frame has the END_HEADERS flag set.

Header compression is stateful. One compression context and one decompression context is used for the entire connection. A decoding error in a header block MUST be treated as a connection error (Section 5.4.1) of type `COMPRESSION_ERROR`.

Each header block is processed as a discrete unit. Header blocks MUST be transmitted as a contiguous sequence of frames, with no interleaved frames of any other type or from any other stream. The last frame in a sequence of HEADERS or CONTINUATION frames has the END_HEADERS flag set. The last frame in a sequence of PUSH_PROMISE or CONTINUATION frames has the END_HEADERS flag set. This allows a header block to be logically equivalent to a single frame.

Header block fragments can only be sent as the payload of HEADERS, PUSH_PROMISE or CONTINUATION frames, because these frames carry data that can modify the compression context maintained by a receiver. An endpoint receiving HEADERS, PUSH_PROMISE or CONTINUATION frames needs to reassemble header blocks and perform decompression even if the frames are to be discarded. A receiver MUST terminate the connection with a connection error (Section 5.4.1) of type `COMPRESSION_ERROR` if it does not decompress a header block.

5. Streams and Multiplexing

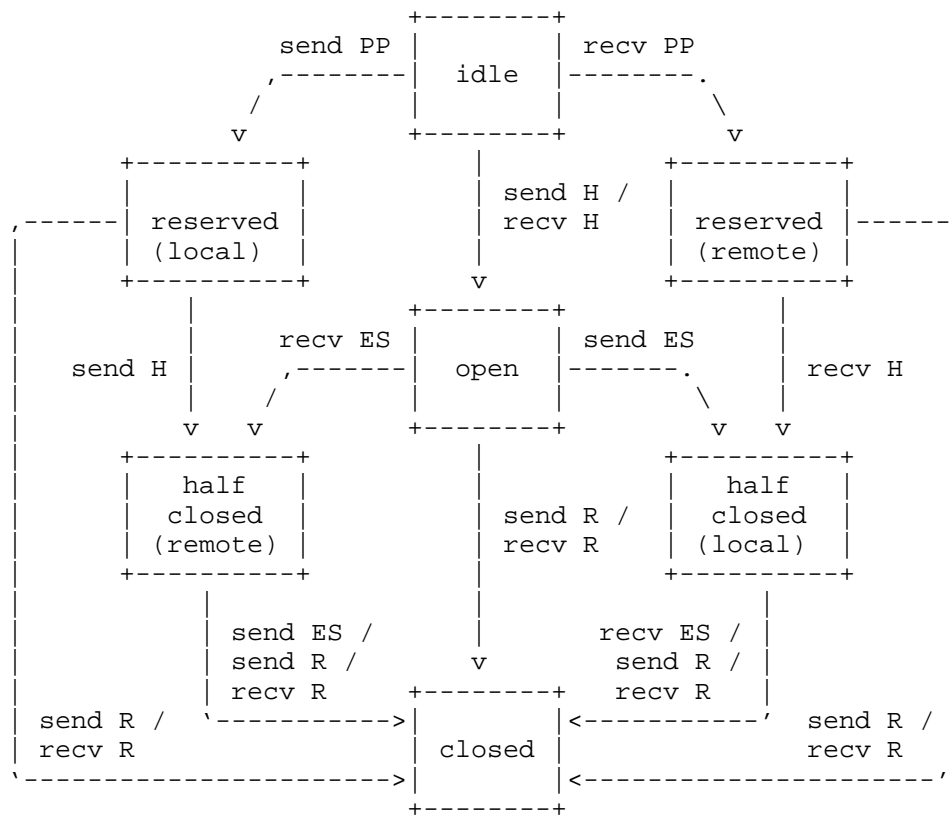
A "stream" is an independent, bi-directional sequence of frames exchanged between the client and server within an HTTP/2 connection. Streams have several important characteristics:

- o A single HTTP/2 connection can contain multiple concurrently open streams, with either endpoint interleaving frames from multiple streams.
- o Streams can be established and used unilaterally or shared by either the client or server.

- o Streams can be closed by either endpoint.
- o The order in which frames are sent on a stream is significant. Recipients process frames in the order they are received. In particular, the order of HEADERS, and DATA frames is semantically significant.
- o Streams are identified by an integer. Stream identifiers are assigned to streams by the endpoint initiating the stream.

5.1. Stream States

The lifecycle of a stream is shown in Figure 2.



send: endpoint sends this frame
 recv: endpoint receives this frame

H: HEADERS frame (with implied CONTINUATIONS)
 PP: PUSH_PROMISE frame (with implied CONTINUATIONS)
 ES: END_STREAM flag
 R: RST_STREAM frame

Figure 2: Stream States

Note that this diagram shows stream state transitions and the frames and flags that affect those transitions only. In this regard, CONTINUATION frames do not result in state transitions; they are effectively part of the HEADERS or PUSH_PROMISE that they follow. For the purpose of state transitions, the END_STREAM flag is processed as a separate event to the frame that bears it; a HEADERS frame with the END_STREAM flag set can cause two state transitions.

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

Streams have the following states:

idle:

All streams start in the "idle" state.

The following transitions are valid from this state:

- * Sending or receiving a HEADERS frame causes the stream to become "open". The stream identifier is selected as described in Section 5.1.1. The same HEADERS frame can also cause a stream to immediately become "half closed".
- * Sending a PUSH_PROMISE frame on another stream reserves the idle stream that is identified for later use. The stream state for the reserved stream transitions to "reserved (local)".
- * Receiving a PUSH_PROMISE frame on another stream reserves an idle stream that is identified for later use. The stream state for the reserved stream transitions to "reserved (remote)".
- * Note that the PUSH_PROMISE frame is not sent on the idle stream, but references the newly reserved stream in the Promised Stream ID field.

Receiving any frame other than HEADERS or PRIORITY on a stream in this state MUST be treated as a connection error (Section 5.4.1) of type `PROTOCOL_ERROR`.

reserved (local):

A stream in the "reserved (local)" state is one that has been promised by sending a PUSH_PROMISE frame. A PUSH_PROMISE frame reserves an idle stream by associating the stream with an open stream that was initiated by the remote peer (see Section 8.2).

In this state, only the following transitions are possible:

- * The endpoint can send a HEADERS frame. This causes the stream to open in a "half closed (remote)" state.
- * Either endpoint can send a RST_STREAM frame to cause the stream to become "closed". This releases the stream reservation.

An endpoint MUST NOT send any type of frame other than HEADERS, RST_STREAM, or PRIORITY in this state.

A PRIORITY or WINDOW_UPDATE frame MAY be received in this state. Receiving any type of frame other than RST_STREAM, PRIORITY or WINDOW_UPDATE on a stream in this state MUST be treated as a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

reserved (remote):

A stream in the "reserved (remote)" state has been reserved by a remote peer.

In this state, only the following transitions are possible:

- * Receiving a HEADERS frame causes the stream to transition to "half closed (local)".
- * Either endpoint can send a RST_STREAM frame to cause the stream to become "closed". This releases the stream reservation.

An endpoint MAY send a PRIORITY frame in this state to reprioritize the reserved stream. An endpoint MUST NOT send any type of frame other than RST_STREAM, WINDOW_UPDATE, or PRIORITY in this state.

Receiving any type of frame other than HEADERS, RST_STREAM or PRIORITY on a stream in this state MUST be treated as a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

open:

A stream in the "open" state may be used by both peers to send frames of any type. In this state, sending peers observe advertised stream level flow control limits (Section 5.2).

From this state either endpoint can send a frame with an END_STREAM flag set, which causes the stream to transition into one of the "half closed" states: an endpoint sending an END_STREAM flag causes the stream state to become "half closed (local)"; an endpoint receiving an END_STREAM flag causes the stream state to become "half closed (remote)".

Either endpoint can send a RST_STREAM frame from this state, causing it to transition immediately to "closed".

half closed (local):

A stream that is in the "half closed (local)" state cannot be used for sending frames other than WINDOW_UPDATE, PRIORITY and RST_STREAM.

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

An endpoint can receive any type of frame in this state. Providing flow control credit using WINDOW_UPDATE frames is necessary to continue receiving flow controlled frames. A receiver can ignore WINDOW_UPDATE frames in this state, which might arrive for a short period after a frame bearing the END_STREAM flag is sent.

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

half closed (remote):

A stream that is "half closed (remote)" is no longer being used by the peer to send frames. In this state, an endpoint is no longer obligated to maintain a receiver flow control window.

If an endpoint receives additional frames for a stream that is in this state, other than WINDOW_UPDATE, PRIORITY or RST_STREAM, it MUST respond with a stream error (Section 5.4.2) of type STREAM_CLOSED.

A stream that is "half closed (remote)" can be used by the endpoint to send frames of any type. In this state, the endpoint continues to observe advertised stream level flow control limits (Section 5.2).

A stream can transition from this state to "closed" by sending a frame that contains an END_STREAM flag, or when either peer sends a RST_STREAM frame.

closed:

The "closed" state is the terminal state.

An endpoint MUST NOT send frames other than PRIORITY on a closed stream. An endpoint that receives any frame other than PRIORITY after receiving a RST_STREAM MUST treat that as a stream error (Section 5.4.2) of type STREAM_CLOSED. Similarly, an endpoint that receives any frames after receiving a frame with the END_STREAM flag set MUST treat that as a connection error (Section 5.4.1) of type STREAM_CLOSED, unless the frame is permitted as described below.

WINDOW_UPDATE or RST_STREAM frames can be received in this state for a short period after a DATA or HEADERS frame containing an END_STREAM flag is sent. Until the remote peer receives and

processes RST_STREAM or the frame bearing the END_STREAM flag, it might send frames of these types. Endpoints MUST ignore WINDOW_UPDATE or RST_STREAM frames received in this state, though endpoints MAY choose to treat frames that arrive a significant time after sending END_STREAM as a connection error (Section 5.4.1) of type `PROTOCOL_ERROR`.

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

If this state is reached as a result of sending a RST_STREAM frame, the peer that receives the RST_STREAM might have already sent - or enqueued for sending - frames on the stream that cannot be withdrawn. An endpoint MUST ignore frames that it receives on closed streams after it has sent a RST_STREAM frame. An endpoint MAY choose to limit the period over which it ignores frames and treat frames that arrive after this time as being in error.

Flow controlled frames (i.e., DATA) received after sending RST_STREAM are counted toward the connection flow control window. Even though these frames might be ignored, because they are sent before the sender receives the RST_STREAM, the sender will consider the frames to count against the flow control window.

An endpoint might receive a PUSH_PROMISE frame after it sends RST_STREAM. PUSH_PROMISE causes a stream to become "reserved" even if the associated stream has been reset. Therefore, a RST_STREAM is needed to close an unwanted promised stream.

In the absence of more specific guidance elsewhere in this document, implementations SHOULD treat the receipt of a frame that is not expressly permitted in the description of a state as a connection error (Section 5.4.1) of type `PROTOCOL_ERROR`. Note that PRIORITY can be sent and received in any stream state. Frames of unknown types are ignored.

An example of the state transitions for an HTTP request/response exchange can be found in Section 8.1. An example of the state transitions for server push can be found in Section 8.2.1 and Section 8.2.2.

5.1.1. Stream Identifiers

Streams are identified with an unsigned 31-bit integer. Streams initiated by a client MUST use odd-numbered stream identifiers; those initiated by the server MUST use even-numbered stream identifiers. A

stream identifier of zero (0x0) is used for connection control messages; the stream identifier zero cannot be used to establish a new stream.

HTTP/1.1 requests that are upgraded to HTTP/2 (see Section 3.2) are responded to with a stream identifier of one (0x1). After the upgrade completes, stream 0x1 is "half closed (local)" to the client. Therefore, stream 0x1 cannot be selected as a new stream identifier by a client that upgrades from HTTP/1.1.

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

The first use of a new stream identifier implicitly closes all streams in the "idle" state that might have been initiated by that peer with a lower-valued stream identifier. For example, if a client sends a HEADERS frame on stream 7 without ever sending a frame on stream 5, then stream 5 transitions to the "closed" state when the first frame for stream 7 is sent or received.

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

5.1.2. Stream Concurrency

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

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

Endpoints **MUST NOT** exceed the limit set by their peer. An endpoint that receives a HEADERS frame that causes their advertised concurrent stream limit to be exceeded **MUST** treat this as a stream error (Section 5.4.2) of type `PROTOCOL_ERROR` or `REFUSED_STREAM`. The choice of error code determines whether the endpoint wishes to enable automatic retry, see Section 8.1.4) for details.

An endpoint that wishes to reduce the value of `SETTINGS_MAX_CONCURRENT_STREAMS` to a value that is below the current number of open streams can either close streams that exceed the new value or allow streams to complete.

5.2. Flow Control

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

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

5.2.1. Flow Control Principles

HTTP/2 stream flow control aims to allow a variety of flow control algorithms to be used without requiring protocol changes. Flow control in HTTP/2 has the following characteristics:

1. Flow control is specific to a connection. Both types of flow control are between the endpoints of a single hop, and not over the entire end-to-end path.
2. Flow control is based on window update frames. Receivers advertise how many octets they are prepared to receive on a stream and for the entire connection. This is a credit-based scheme.
3. Flow control is directional with overall control provided by the receiver. A receiver **MAY** choose to set any window size that it desires for each stream and for the entire connection. A sender **MUST** respect flow control limits imposed by a receiver. Clients, servers and intermediaries all independently advertise their flow control window as a receiver and abide by the flow control limits set by their peer when sending.
4. The initial value for the flow control window is 65,535 octets for both new streams and the overall connection.

5. The frame type determines whether flow control applies to a frame. Of the frames specified in this document, only DATA frames are subject to flow control; all other frame types do not consume space in the advertised flow control window. This ensures that important control frames are not blocked by flow control.
6. Flow control cannot be disabled.
7. HTTP/2 defines only the format and semantics of the WINDOW_UPDATE frame (Section 6.9). This document does not stipulate how a receiver decides when to send this frame or the value that it sends, nor does it specify how a sender chooses to send packets. Implementations are able to select any algorithm that suits their needs.

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

5.2.2. Appropriate Use of Flow Control

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

Deployments that do not require this capability can advertise a flow control window of the maximum size ($2^{31}-1$), and by maintaining this window by sending a WINDOW_UPDATE frame when any data is received. This effectively disables flow control for that receiver. Conversely, a sender is always subject to the flow control window advertised by the receiver.

Deployments with constrained resources (for example, memory) can employ flow control to limit the amount of memory a peer can consume. Note, however, that this can lead to suboptimal use of available network resources if flow control is enabled without knowledge of the bandwidth-delay product (see [RFC7323]).

Even with full awareness of the current bandwidth-delay product, implementation of flow control can be difficult. When using flow control, the receiver **MUST** read from the TCP receive buffer in a

timely fashion. Failure to do so could lead to a deadlock when critical frames, such as WINDOW_UPDATE, are not read and acted upon.

5.3. Stream priority

A client can assign a priority for a new stream by including prioritization information in the HEADERS frame (Section 6.2) that opens the stream. At any other time, the PRIORITY frame (Section 6.3) can be used to change the priority of a stream.

The purpose of prioritization is to allow an endpoint to express how it would prefer its peer allocate resources when managing concurrent streams. Most importantly, priority can be used to select streams for transmitting frames when there is limited capacity for sending.

Streams can be prioritized by marking them as dependent on the completion of other streams (Section 5.3.1). Each dependency is assigned a relative weight, a number that is used to determine the relative proportion of available resources that are assigned to streams dependent on the same stream.

Explicitly setting the priority for a stream is input to a prioritization process. It does not guarantee any particular processing or transmission order for the stream relative to any other stream. An endpoint cannot force a peer to process concurrent streams in a particular order using priority. Expressing priority is therefore only ever a suggestion.

Prioritization information can be omitted from messages. Defaults are used prior to any explicit values being provided (Section 5.3.5).

5.3.1. Stream Dependencies

Each stream can be given an explicit dependency on another stream. Including a dependency expresses a preference to allocate resources to the identified stream rather than to the dependent stream.

A stream that is not dependent on any other stream is given a stream dependency of 0x0. In other words, the non-existent stream 0 forms the root of the tree.

A stream that depends on another stream is a dependent stream. The stream upon which a stream is dependent is a parent stream. A dependency on a stream that is not currently in the tree - such as a stream in the "idle" state - results in that stream being given a default priority (Section 5.3.5).

When assigning a dependency on another stream, the stream is added as a new dependency of the parent stream. Dependent streams that share the same parent are not ordered with respect to each other. For example, if streams B and C are dependent on stream A, and if stream D is created with a dependency on stream A, this results in a dependency order of A followed by B, C, and D in any order.

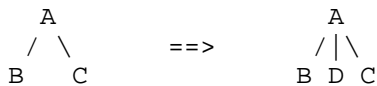


Figure 3: Example of Default Dependency Creation

An exclusive flag allows for the insertion of a new level of dependencies. The exclusive flag causes the stream to become the sole dependency of its parent stream, causing other dependencies to become dependent on the exclusive stream. In the previous example, if stream D is created with an exclusive dependency on stream A, this results in D becoming the dependency parent of B and C.



Figure 4: Example of Exclusive Dependency Creation

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

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

5.3.2. Dependency Weighting

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

Streams with the same parent **SHOULD** be allocated resources proportionally based on their weight. Thus, if stream B depends on stream A with weight 4, and C depends on stream A with weight 12, and if no progress can be made on A, stream B ideally receives one third of the resources allocated to stream C.

5.3.3. Reprioritization

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

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

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

For example, consider an original dependency tree where B and C depend on A, D and E depend on C, and F depends on D. If A is made dependent on D, then D takes the place of A. All other dependency relationships stay the same, except for F, which becomes dependent on A if the reprioritization is exclusive.

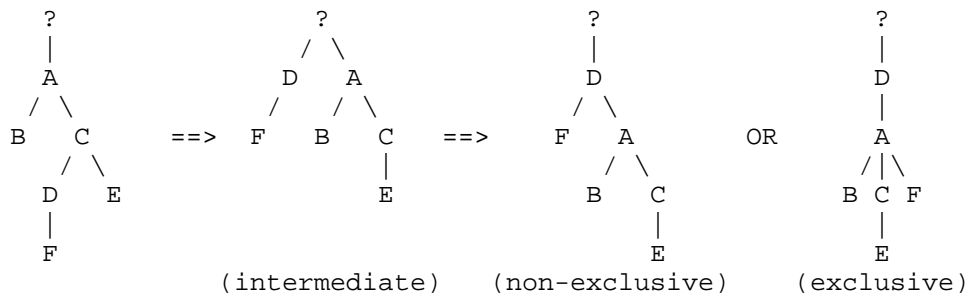


Figure 5: Example of Dependency Reordering

5.3.4. Prioritization State Management

When a stream is removed from the dependency tree, its dependencies can be moved to become dependent on the parent of the closed stream. The weights of new dependencies are recalculated by distributing the weight of the dependency of the closed stream proportionally based on the weights of its dependencies.

Streams that are removed from the dependency tree cause some prioritization information to be lost. Resources are shared between streams with the same parent stream, which means that if a stream in that set closes or becomes blocked, any spare capacity allocated to a stream is distributed to the immediate neighbors of the stream.

However, if the common dependency is removed from the tree, those streams share resources with streams at the next highest level.

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

It is possible for a stream to become closed while prioritization information that creates a dependency on that stream is in transit. If a stream identified in a dependency has no associated priority information, then the dependent stream is instead assigned a default priority (Section 5.3.5). This potentially creates suboptimal prioritization, since the stream could be given a priority that is different to what is intended.

To avoid these problems, an endpoint **SHOULD** retain stream prioritization state for a period after streams become closed. The longer state is retained, the lower the chance that streams are assigned incorrect or default priority values.

Similarly, streams that are in the "idle" state can be assigned priority or become a parent of other streams. This allows for the creation of a grouping node in the dependency tree, which enables more flexible expressions of priority. Idle streams begin with a default priority (Section 5.3.5).

The retention of priority information for streams that are not counted toward the limit set by `SETTINGS_MAX_CONCURRENT_STREAMS` could create a large state burden for an endpoint. Therefore the amount of prioritization state that is retained **MAY** be limited.

The amount of additional state an endpoint maintains for prioritization could be dependent on load; under high load, prioritization state can be discarded to limit resource commitments. In extreme cases, an endpoint could even discard prioritization state for active or reserved streams. If a limit is applied, endpoints **SHOULD** maintain state for at least as many streams as allowed by their setting for `SETTINGS_MAX_CONCURRENT_STREAMS`. Implementations **SHOULD** also attempt to retain state for streams that are in active use in the priority tree.

An endpoint receiving a PRIORITY frame that changes the priority of a closed stream SHOULD alter the dependencies of the streams that depend on it, if it has retained enough state to do so.

5.3.5. Default Priorities

All streams are initially assigned a non-exclusive dependency on stream 0x0. Pushed streams (Section 8.2) initially depend on their associated stream. In both cases, streams are assigned a default weight of 16.

5.4. Error Handling

HTTP/2 framing permits two classes of error:

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

A list of error codes is included in Section 7.

5.4.1. Connection Error Handling

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

An endpoint that encounters a connection error SHOULD first send a GOAWAY frame (Section 6.8) with the stream identifier of the last stream that it successfully received from its peer. The GOAWAY frame includes an error code that indicates why the connection is terminating. After sending the GOAWAY frame for an error condition, the endpoint MUST close the TCP connection.

It is possible that the GOAWAY will not be reliably received by the receiving endpoint (see [RFC7230], Section 6.6). In the event of a connection error, GOAWAY only provides a best effort attempt to communicate with the peer about why the connection is being terminated.

An endpoint can end a connection at any time. In particular, an endpoint MAY choose to treat a stream error as a connection error. Endpoints SHOULD send a GOAWAY frame when ending a connection, providing that circumstances permit it.

5.4.2. Stream Error Handling

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

An endpoint that detects a stream error sends a RST_STREAM frame (Section 6.4) that contains the stream identifier of the stream where the error occurred. The RST_STREAM frame includes an error code that indicates the type of error.

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

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

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

5.4.3. Connection Termination

If the TCP connection is closed or reset while streams remain in open or half closed states, then the affected streams cannot be automatically retried (see Section 8.1.4 for details).

5.5. Extending HTTP/2

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

This applies to the protocol elements defined in this document. This does not affect the existing options for extending HTTP, such as defining new methods, status codes, or header fields.

Extensions are permitted to use new frame types (Section 4.1), new settings (Section 6.5.2), or new error codes (Section 7). Registries are established for managing these extension points: frame types

(Section 11.2), settings (Section 11.3) and error codes (Section 11.4).

Implementations MUST ignore unknown or unsupported values in all extensible protocol elements. Implementations MUST discard frames that have unknown or unsupported types. This means that any of these extension points can be safely used by extensions without prior arrangement or negotiation. However, extension frames that appear in the middle of a header block (Section 4.3) are not permitted; these MUST be treated as a connection error (Section 5.4.1) of type `PROTOCOL_ERROR`.

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

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

6. Frame Definitions

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

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

6.1. DATA

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

DATA frames MAY also contain padding. Padding can be added to DATA frames to obscure the size of messages. Padding is a security feature; see Section 10.7.

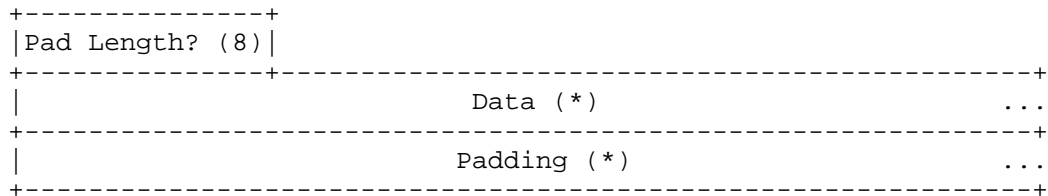


Figure 6: DATA Frame Payload

The DATA frame contains the following fields:

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

Data: Application data. The amount of data is the remainder of the frame payload after subtracting the length of the other fields that are present.

Padding: Padding octets that contain no application semantic value. Padding octets MUST be set to zero when sending. A receiver is not obligated to verify padding, but MAY treat non-zero padding as a connection error (Section 5.4.1) of type `PROTOCOL_ERROR`.

The DATA frame defines the following flags:

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

PADDED (0x8): Bit 3 being set indicates that the Pad Length field and any padding that it describes is present.

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

DATA frames are subject to flow control and can only be sent when a stream is in the "open" or "half closed (remote)" states. The entire DATA frame payload is included in flow control, including Pad Length and Padding fields if present. If a DATA frame is received whose

stream is not in "open" or "half closed (local)" state, the recipient MUST respond with a stream error (Section 5.4.2) of type `STREAM_CLOSED`.

The total number of padding octets is determined by the value of the Pad Length field. If the length of the padding is the length of the frame payload or greater, the recipient MUST treat this as a connection error (Section 5.4.1) of type `PROTOCOL_ERROR`.

Note: A frame can be increased in size by one octet by including a Pad Length field with a value of zero.

6.2. HEADERS

The HEADERS frame (type=0x1) is used to open a stream (Section 5.1), and additionally carries a header block fragment. HEADERS frames can be sent on a stream in the "open" or "half closed (remote)" states.

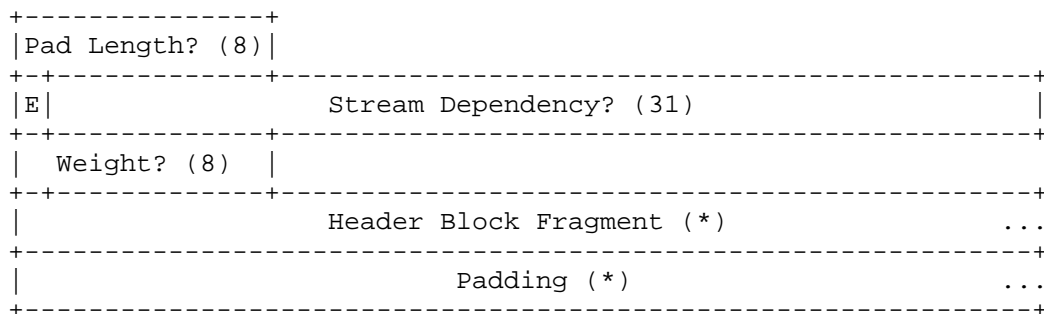


Figure 7: HEADERS Frame Payload

The HEADERS frame payload has the following fields:

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

E: A single bit flag indicates that the stream dependency is exclusive, see Section 5.3. This field is only present if the `PRIORITY` flag is set.

Stream Dependency: A 31-bit stream identifier for the stream that this stream depends on, see Section 5.3. This field is only present if the `PRIORITY` flag is set.

Weight: An unsigned 8-bit integer representing a priority weight for the stream, see Section 5.3. Add one to the value to obtain a

weight between 1 and 256. This field is only present if the `PRIORITY` flag is set.

Header Block Fragment: A header block fragment (Section 4.3).

Padding: Padding octets.

The HEADERS frame defines the following flags:

`END_STREAM` (0x1): Bit 0 being set indicates that the header block (Section 4.3) is the last that the endpoint will send for the identified stream.

A HEADERS frame carries the `END_STREAM` flag that signals the end of a stream. However, a HEADERS frame with the `END_STREAM` flag set can be followed by `CONTINUATION` frames on the same stream. Logically, the `CONTINUATION` frames are part of the HEADERS frame.

`END_HEADERS` (0x4): Bit 2 being set indicates that this frame contains an entire header block (Section 4.3) and is not followed by any `CONTINUATION` frames.

A HEADERS frame without the `END_HEADERS` flag set **MUST** be followed by a `CONTINUATION` frame for the same stream. A receiver **MUST** treat the receipt of any other type of frame or a frame on a different stream as a connection error (Section 5.4.1) of type `PROTOCOL_ERROR`.

`PADDED` (0x8): Bit 3 being set indicates that the Pad Length field and any padding that it describes is present.

`PRIORITY` (0x20): Bit 5 being set indicates that the Exclusive Flag (E), Stream Dependency, and Weight fields are present; see Section 5.3.

The payload of a HEADERS frame contains a header block fragment (Section 4.3). A header block that does not fit within a HEADERS frame is continued in a `CONTINUATION` frame (Section 6.10).

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

The HEADERS frame changes the connection state as described in Section 4.3.

The HEADERS frame can include padding. Padding fields and flags are identical to those defined for DATA frames (Section 6.1).

Prioritization information in a HEADERS frame is logically equivalent to a separate PRIORITY frame, but inclusion in HEADERS avoids the potential for churn in stream prioritization when new streams are created. Prioritization fields in HEADERS frames subsequent to the first on a stream reprioritize the stream (Section 5.3.3).

6.3. PRIORITY

The PRIORITY frame (type=0x2) specifies the sender-advised priority of a stream (Section 5.3). It can be sent at any time for any stream, including idle or closed streams.

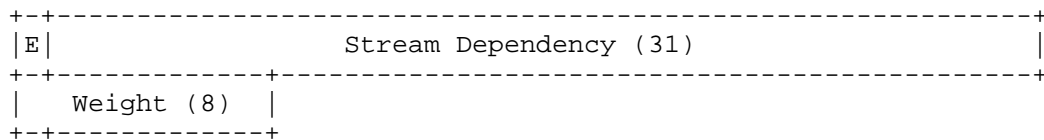


Figure 8: PRIORITY Frame Payload

The payload of a PRIORITY frame contains the following fields:

E: A single bit flag indicates that the stream dependency is exclusive, see Section 5.3.

Stream Dependency: A 31-bit stream identifier for the stream that this stream depends on, see Section 5.3.

Weight: An unsigned 8-bit integer representing a priority weight for the stream, see Section 5.3. Add one to the value to obtain a weight between 1 and 256.

The PRIORITY frame does not define any flags.

The PRIORITY frame always identifies a stream. If a PRIORITY frame is received with a stream identifier of 0x0, the recipient **MUST** respond with a connection error (Section 5.4.1) of type `PROTOCOL_ERROR`.

The PRIORITY frame can be sent on a stream in any state, though it cannot be sent between consecutive frames that comprise a single header block (Section 4.3). Note that this frame could arrive after processing or frame sending has completed, which would cause it to have no effect on the identified stream. For a stream that is in the "half closed (remote)" or "closed" - state, this frame can only

affect processing of the identified stream and its dependent streams and not frame transmission on that stream.

The PRIORITY frame can be sent for a stream in the "idle" or "closed" states. This allows for the reprioritization of a group of dependent streams by altering the priority of an unused or closed parent stream.

A PRIORITY frame with a length other than 5 octets MUST be treated as a stream error (Section 5.4.2) of type FRAME_SIZE_ERROR.

6.4. RST_STREAM

The RST_STREAM frame (type=0x3) allows for immediate termination of a stream. RST_STREAM is sent to request cancellation of a stream, or to indicate that an error condition has occurred.



Figure 9: RST_STREAM Frame Payload

The RST_STREAM frame contains a single unsigned, 32-bit integer identifying the error code (Section 7). The error code indicates why the stream is being terminated.

The RST_STREAM frame does not define any flags.

The RST_STREAM frame fully terminates the referenced stream and causes it to enter the closed state. After receiving a RST_STREAM on a stream, the receiver MUST NOT send additional frames for that stream, with the exception of PRIORITY. However, after sending the RST_STREAM, the sending endpoint MUST be prepared to receive and process additional frames sent on the stream that might have been sent by the peer prior to the arrival of the RST_STREAM.

RST_STREAM frames MUST be associated with a stream. If a RST_STREAM frame is received with a stream identifier of 0x0, the recipient MUST treat this as a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

RST_STREAM frames MUST NOT be sent for a stream in the "idle" state. If a RST_STREAM frame identifying an idle stream is received, the recipient MUST treat this as a connection error (Section 5.4.1) of type PROTOCOL_ERROR.

A RST_STREAM frame with a length other than 4 octets MUST be treated as a connection error (Section 5.4.1) of type FRAME_SIZE_ERROR.

6.5. SETTINGS

The SETTINGS frame (type=0x4) conveys configuration parameters that affect how endpoints communicate, such as preferences and constraints on peer behavior. The SETTINGS frame is also used to acknowledge the receipt of those parameters. Individually, a SETTINGS parameter can also be referred to as a "setting".

SETTINGS parameters are not negotiated; they describe characteristics of the sending peer, which are used by the receiving peer. Different values for the same parameter can be advertised by each peer. For example, a client might set a high initial flow control window, whereas a server might set a lower value to conserve resources.

A SETTINGS frame MUST be sent by both endpoints at the start of a connection, and MAY be sent at any other time by either endpoint over the lifetime of the connection. Implementations MUST support all of the parameters defined by this specification.

Each parameter in a SETTINGS frame replaces any existing value for that parameter. Parameters are processed in the order in which they appear, and a receiver of a SETTINGS frame does not need to maintain any state other than the current value of its parameters. Therefore, the value of a SETTINGS parameter is the last value that is seen by a receiver.

SETTINGS parameters are acknowledged by the receiving peer. To enable this, the SETTINGS frame defines the following flag:

ACK (0x1): Bit 0 being set indicates that this frame acknowledges receipt and application of the peer's SETTINGS frame. When this bit is set, the payload of the SETTINGS frame MUST be empty. Receipt of a SETTINGS frame with the ACK flag set and a length field value other than 0 MUST be treated as a connection error (Section 5.4.1) of type FRAME_SIZE_ERROR. For more info, see Settings Synchronization (Section 6.5.3).

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

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

A SETTINGS frame with a length other than a multiple of 6 octets MUST be treated as a connection error (Section 5.4.1) of type `FRAME_SIZE_ERROR`.

6.5.1. SETTINGS Format

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

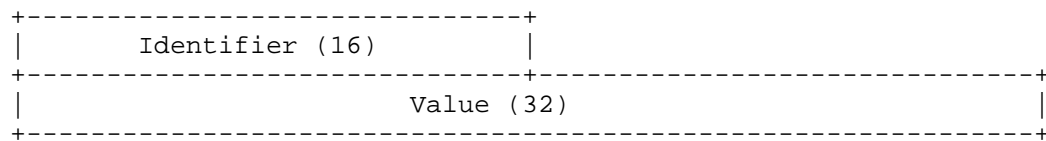


Figure 10: Setting Format

6.5.2. Defined SETTINGS Parameters

The following parameters are defined:

`SETTINGS_HEADER_TABLE_SIZE (0x1)`: Allows the sender to inform the remote endpoint of the maximum size of the header compression table used to decode header blocks, in octets. The encoder can select any size equal to or less than this value by using signaling specific to the header compression format inside a header block, see [COMPRESSION]. The initial value is 4,096 octets.

`SETTINGS_ENABLE_PUSH (0x2)`: This setting can be use to disable server push (Section 8.2). An endpoint MUST NOT send a `PUSH_PROMISE` frame if it receives this parameter set to a value of 0. An endpoint that has both set this parameter to 0 and had it acknowledged MUST treat the receipt of a `PUSH_PROMISE` frame as a connection error (Section 5.4.1) of type `PROTOCOL_ERROR`.

The initial value is 1, which indicates that server push is permitted. Any value other than 0 or 1 MUST be treated as a connection error (Section 5.4.1) of type `PROTOCOL_ERROR`.

`SETTINGS_MAX_CONCURRENT_STREAMS (0x3)`: Indicates the maximum number of concurrent streams that the sender will allow. This limit is directional: it applies to the number of streams that the sender

permits the receiver to create. Initially there is no limit to this value. It is recommended that this value be no smaller than 100, so as to not unnecessarily limit parallelism.

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

`SETTINGS_INITIAL_WINDOW_SIZE` (0x4): Indicates the sender's initial window size (in octets) for stream level flow control. The initial value is $2^{16}-1$ (65,535) octets.

This setting affects the window size of all streams, see Section 6.9.2.

Values above the maximum flow control window size of $2^{31}-1$ MUST be treated as a connection error (Section 5.4.1) of type `FLOW_CONTROL_ERROR`.

`SETTINGS_MAX_FRAME_SIZE` (0x5): Indicates the size of the largest frame payload that the sender is willing to receive, in octets.

The initial value is 2^{14} (16,384) octets. The value advertised by an endpoint MUST be between this initial value and the maximum allowed frame size ($2^{24}-1$ or 16,777,215 octets), inclusive. Values outside this range MUST be treated as a connection error (Section 5.4.1) of type `PROTOCOL_ERROR`.

`SETTINGS_MAX_HEADER_LIST_SIZE` (0x6): This advisory setting informs a peer of the maximum size of header list that the sender is prepared to accept, in octets. The value is based on the uncompressed size of header fields, including the length of the name and value in octets plus an overhead of 32 octets for each header field.

For any given request, a lower limit than what is advertised MAY be enforced. The initial value of this setting is unlimited.

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

6.5.3. Settings Synchronization

Most values in SETTINGS benefit from or require an understanding of when the peer has received and applied the changed parameter values. In order to provide such synchronization timepoints, the recipient of a SETTINGS frame in which the ACK flag is not set MUST apply the updated parameters as soon as possible upon receipt.

The values in the SETTINGS frame MUST be processed in the order they appear, with no other frame processing between values. Unsupported parameters MUST be ignored. Once all values have been processed, the recipient MUST immediately emit a SETTINGS frame with the ACK flag set. Upon receiving a SETTINGS frame with the ACK flag set, the sender of the altered parameters can rely on the setting having been applied.

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

6.6. PUSH_PROMISE

The PUSH_PROMISE frame (type=0x5) is used to notify the peer endpoint in advance of streams the sender intends to initiate. The PUSH_PROMISE frame includes the unsigned 31-bit identifier of the stream the endpoint plans to create along with a set of headers that provide additional context for the stream. Section 8.2 contains a thorough description of the use of PUSH_PROMISE frames.

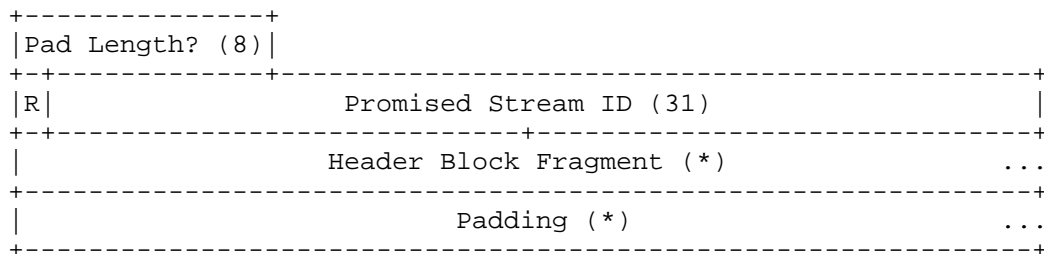


Figure 11: PUSH_PROMISE Payload Format

The PUSH_PROMISE frame payload has the following fields:

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

R: A single reserved bit.

Promised Stream ID: An unsigned 31-bit integer that identifies the stream that is reserved by the PUSH_PROMISE. The promised stream identifier **MUST** be a valid choice for the next stream sent by the sender (see new stream identifier (Section 5.1.1)).

Header Block Fragment: A header block fragment (Section 4.3) containing request header fields.

Padding: Padding octets.

The PUSH_PROMISE frame defines the following flags:

END_HEADERS (0x4): Bit 2 being set indicates that this frame contains an entire header block (Section 4.3) and is not followed by any CONTINUATION frames.

A PUSH_PROMISE frame without the END_HEADERS flag set **MUST** be followed by a CONTINUATION frame for the same stream. A receiver **MUST** treat the receipt of any other type of frame or a frame on a different stream as a connection error (Section 5.4.1) of type `PROTOCOL_ERROR`.

PADDED (0x8): Bit 3 being set indicates that the Pad Length field and any padding that it describes is present.

PUSH_PROMISE frames **MUST** be associated with a peer-initiated stream that is in either the "open" or "half closed (remote)" state. The stream identifier of a PUSH_PROMISE frame indicates the stream it is associated with. If the stream identifier field specifies the value 0x0, a recipient **MUST** respond with a connection error (Section 5.4.1) of type `PROTOCOL_ERROR`.

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

PUSH_PROMISE **MUST NOT** be sent if the `SETTINGS_ENABLE_PUSH` setting of the peer endpoint is set to 0. An endpoint that has set this setting and has received acknowledgement **MUST** treat the receipt of a PUSH_PROMISE frame as a connection error (Section 5.4.1) of type `PROTOCOL_ERROR`.

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

A PUSH_PROMISE frame modifies the connection state in two ways. The inclusion of a header block (Section 4.3) potentially modifies the

state maintained for header compression. PUSH_PROMISE also reserves a stream for later use, causing the promised stream to enter the "reserved" state. A sender MUST NOT send a PUSH_PROMISE on a stream unless that stream is either "open" or "half closed (remote)"; the sender MUST ensure that the promised stream is a valid choice for a new stream identifier (Section 5.1.1) (that is, the promised stream MUST be in the "idle" state).

Since `PUSH_PROMISE` reserves a stream, ignoring a `PUSH_PROMISE` frame causes the stream state to become indeterminate. A receiver **MUST** treat the receipt of a `PUSH_PROMISE` on a stream that is neither "open" nor "half closed (local)" as a connection error (Section 5.4.1) of type `PROTOCOL_ERROR`. However, an endpoint that has sent `RST_STREAM` on the associated stream **MUST** handle `PUSH_PROMISE` frames that might have been created before the `RST_STREAM` frame is received and processed.

A receiver MUST treat the receipt of a PUSH_PROMISE that promises an illegal stream identifier (Section 5.1.1) (that is, an identifier for a stream that is not currently in the "idle" state) as a connection error (Section 5.4.1) of type `PROTOCOL_ERROR`.

The `PUSH_PROMISE` frame can include padding. Padding fields and flags are identical to those defined for `DATA` frames (Section 6.1).

6.7. PING

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

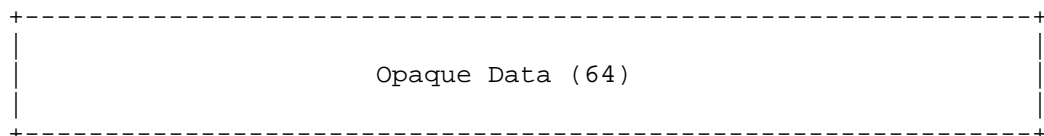


Figure 12: PING Payload Format

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

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

The PING frame defines the following flags:

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

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

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

6.8. GOAWAY

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

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

There is an inherent race condition between an endpoint starting new streams and the remote sending a GOAWAY frame. To deal with this case, the GOAWAY contains the stream identifier of the last peer-initiated stream which was or might be processed on the sending endpoint in this connection. For instance, if the server sends a GOAWAY frame, the identified stream is the highest numbered stream initiated by the client.

If the receiver of the GOAWAY has sent data on streams with a higher stream identifier than what is indicated in the GOAWAY frame, those streams are not or will not be processed. The receiver of the GOAWAY frame can treat the streams as though they had never been created at all, thereby allowing those streams to be retried later on a new connection.

Endpoints **SHOULD** always send a GOAWAY frame before closing a connection so that the remote peer can know whether a stream has been partially processed or not. For example, if an HTTP client sends a

POST at the same time that a server closes a connection, the client cannot know if the server started to process that POST request if the server does not send a GOAWAY frame to indicate what streams it might have acted on.

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

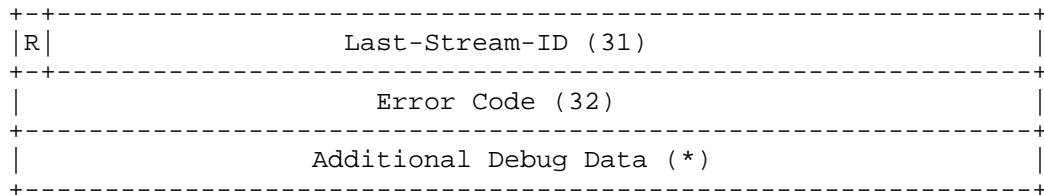


Figure 13: GOAWAY Payload Format

The GOAWAY frame does not define any flags.

The GOAWAY frame applies to the connection, not a specific stream. An endpoint **MUST** treat a GOAWAY frame with a stream identifier other than 0x0 as a connection error (Section 5.4.1) of type `PROTOCOL_ERROR`.

The last stream identifier in the GOAWAY frame contains the highest numbered stream identifier for which the sender of the GOAWAY frame might have taken some action on, or might yet take action on. All streams up to and including the identified stream might have been processed in some way. The last stream identifier can be set to 0 if no streams were processed.

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

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

On streams with lower or equal numbered identifiers that were not closed completely prior to the connection being closed, re-attempting requests, transactions, or any protocol activity is not possible, with the exception of idempotent actions like HTTP GET, PUT, or DELETE. Any protocol activity that uses higher numbered streams can be safely retried using a new connection.

Activity on streams numbered lower or equal to the last stream identifier might still complete successfully. The sender of a GOAWAY

frame might gracefully shut down a connection by sending a GOAWAY frame, maintaining the connection in an open state until all in-progress streams complete.

An endpoint MAY send multiple GOAWAY frames if circumstances change. For instance, an endpoint that sends GOAWAY with NO_ERROR during graceful shutdown could subsequently encounter a condition that requires immediate termination of the connection. The last stream identifier from the last GOAWAY frame received indicates which streams could have been acted upon. Endpoints MUST NOT increase the value they send in the last stream identifier, since the peers might already have retried unprocessed requests on another connection.

A client that is unable to retry requests loses all requests that are in flight when the server closes the connection. This is especially true for intermediaries that might not be serving clients using HTTP/2. A server that is attempting to gracefully shut down a connection SHOULD send an initial GOAWAY frame with the last stream identifier set to $2^{31}-1$ and a NO_ERROR code. This signals to the client that a shutdown is imminent and that no further requests can be initiated. After waiting at least one round trip time, the server can send another GOAWAY frame with an updated last stream identifier. This ensures that a connection can be cleanly shut down without losing requests.

After sending a GOAWAY frame, the sender can discard frames for streams with identifiers higher than the identified last stream. However, any frames that alter connection state cannot be completely ignored. For instance, HEADERS, PUSH_PROMISE and CONTINUATION frames MUST be minimally processed to ensure the state maintained for header compression is consistent (see Section 4.3); similarly DATA frames MUST be counted toward the connection flow control window. Failure to process these frames can cause flow control or header compression state to become unsynchronized.

The GOAWAY frame also contains a 32-bit error code (Section 7) that contains the reason for closing the connection.

Endpoints MAY append opaque data to the payload of any GOAWAY frame. Additional debug data is intended for diagnostic purposes only and carries no semantic value. Debug information could contain security- or privacy-sensitive data. Logged or otherwise persistently stored debug data MUST have adequate safeguards to prevent unauthorized access.

6.9. WINDOW_UPDATE

The WINDOW_UPDATE frame (type=0x8) is used to implement flow control; see Section 5.2 for an overview.

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

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

Flow control only applies to frames that are identified as being subject to flow control. Of the frame types defined in this document, this includes only DATA frames. Frames that are exempt from flow control MUST be accepted and processed, unless the receiver is unable to assign resources to handling the frame. A receiver MAY respond with a stream error (Section 5.4.2) or connection error (Section 5.4.1) of type FLOW_CONTROL_ERROR if it is unable to accept a frame.

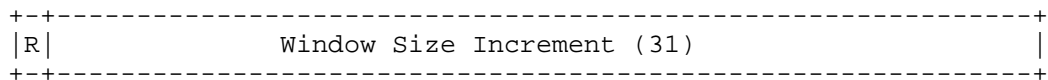


Figure 14: WINDOW_UPDATE Payload Format

The payload of a WINDOW_UPDATE frame is one reserved bit, plus an unsigned 31-bit integer indicating the number of octets that the sender can transmit in addition to the existing flow control window. The legal range for the increment to the flow control window is 1 to $2^{31}-1$ (2,147,483,647) octets.

The WINDOW_UPDATE frame does not define any flags.

The WINDOW_UPDATE frame can be specific to a stream or to the entire connection. In the former case, the frame's stream identifier indicates the affected stream; in the latter, the value "0" indicates that the entire connection is the subject of the frame.

A receiver MUST treat the receipt of a WINDOW_UPDATE frame with a flow control window increment of 0 as a stream error (Section 5.4.2) of type PROTOCOL_ERROR; errors on the connection flow control window MUST be treated as a connection error (Section 5.4.1).

WINDOW_UPDATE can be sent by a peer that has sent a frame bearing the END_STREAM flag. This means that a receiver could receive a WINDOW_UPDATE frame on a "half closed (remote)" or "closed" stream. A receiver MUST NOT treat this as an error, see Section 5.1.

A receiver that receives a flow controlled frame MUST always account for its contribution against the connection flow control window, unless the receiver treats this as a connection error (Section 5.4.1). This is necessary even if the frame is in error. Since the sender counts the frame toward the flow control window, if the receiver does not, the flow control window at sender and receiver can become different.

A WINDOW_UPDATE frame with a length other than 4 octets MUST be treated as a connection error (Section 5.4.1) of type FRAME_SIZE_ERROR.

6.9.1. The Flow Control Window

Flow control in HTTP/2 is implemented using a window kept by each sender on every stream. The flow control window is a simple integer value that indicates how many octets of data the sender is permitted to transmit; as such, its size is a measure of the buffering capacity of the receiver.

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

For flow control calculations, the 9 octet frame header is not counted.

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

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

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

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

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

6.9.2. Initial Flow Control Window Size

When an HTTP/2 connection is first established, new streams are created with an initial flow control window size of 65,535 octets. The connection flow control window is 65,535 octets. Both endpoints can adjust the initial window size for new streams by including a value for SETTINGS_INITIAL_WINDOW_SIZE in the SETTINGS frame that forms part of the connection preface. The connection flow control window can only be changed using WINDOW_UPDATE frames.

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

A SETTINGS frame can alter the initial flow control window size for all streams in the "open" or "half closed (remote)" state. When the value of SETTINGS_INITIAL_WINDOW_SIZE changes, a receiver MUST adjust the size of all stream flow control windows that it maintains by the difference between the new value and the old value.

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

For example, if the client sends 60KB immediately on connection establishment, and the server sets the initial window size to be 16KB, the client will recalculate the available flow control window to be -44KB on receipt of the SETTINGS frame. The client retains a negative flow control window until WINDOW_UPDATE frames restore the window to being positive, after which the client can resume sending.

A SETTINGS frame cannot alter the connection flow control window.

An endpoint **MUST** treat a change to `SETTINGS_INITIAL_WINDOW_SIZE` that causes any flow control window to exceed the maximum size as a connection error (Section 5.4.1) of type `FLOW_CONTROL_ERROR`.

6.9.3. Reducing the Stream Window Size

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

After sending a `SETTINGS` frame that reduces the initial flow control window size, a receiver **MAY** continue to process streams that exceed flow control limits. Allowing streams to continue does not allow the receiver to immediately reduce the space it reserves for flow control windows. Progress on these streams can also stall, since `WINDOW_UPDATE` frames are needed to allow the sender to resume sending. The receiver **MAY** instead send a `RST_STREAM` with `FLOW_CONTROL_ERROR` error code for the affected streams.

6.10. CONTINUATION

The `CONTINUATION` frame (type=0x9) is used to continue a sequence of header block fragments (Section 4.3). Any number of `CONTINUATION` frames can be sent, as long as the preceding frame is on the same stream and is a `HEADERS`, `PUSH_PROMISE` or `CONTINUATION` frame without the `END_HEADERS` flag set.

```
+-----+
|               Header Block Fragment (*)               ...
+-----+
```

Figure 15: `CONTINUATION` Frame Payload

The `CONTINUATION` frame payload contains a header block fragment (Section 4.3).

The `CONTINUATION` frame defines the following flag:

`END_HEADERS` (0x4): Bit 2 being set indicates that this frame ends a header block (Section 4.3).

If the `END_HEADERS` bit is not set, this frame **MUST** be followed by another `CONTINUATION` frame. A receiver **MUST** treat the receipt of any other type of frame or a frame on a different stream as a connection error (Section 5.4.1) of type `PROTOCOL_ERROR`.

The CONTINUATION frame changes the connection state as defined in Section 4.3.

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

A CONTINUATION frame MUST be preceded by a `HEADERS`, `PUSH_PROMISE` or CONTINUATION frame without the `END_HEADERS` flag set. A recipient that observes violation of this rule MUST respond with a connection error (Section 5.4.1) of type `PROTOCOL_ERROR`.

7. Error Codes

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

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

The following error codes are defined:

`NO_ERROR` (0x0): The associated condition is not as a result of an error. For example, a `GOAWAY` might include this code to indicate graceful shutdown of a connection.

`PROTOCOL_ERROR` (0x1): The endpoint detected an unspecific protocol error. This error is for use when a more specific error code is not available.

`INTERNAL_ERROR` (0x2): The endpoint encountered an unexpected internal error.

`FLOW_CONTROL_ERROR` (0x3): The endpoint detected that its peer violated the flow control protocol.

`SETTINGS_TIMEOUT` (0x4): The endpoint sent a `SETTINGS` frame, but did not receive a response in a timely manner. See `Settings Synchronization` (Section 6.5.3).

`STREAM_CLOSED` (0x5): The endpoint received a frame after a stream was half closed.

`FRAME_SIZE_ERROR` (0x6): The endpoint received a frame with an invalid size.

REFUSED_STREAM (0x7): The endpoint refuses the stream prior to performing any application processing, see Section 8.1.4 for details.

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

COMPRESSION_ERROR (0x9): The endpoint is unable to maintain the header compression context for the connection.

CONNECT_ERROR (0xa): The connection established in response to a CONNECT request (Section 8.3) was reset or abnormally closed.

ENHANCE_YOUR_CALM (0xb): The endpoint detected that its peer is exhibiting a behavior that might be generating excessive load.

INADEQUATE_SECURITY (0xc): The underlying transport has properties that do not meet minimum security requirements (see Section 9.2).

HTTP_1_1_REQUIRED (0xd): The endpoint requires that HTTP/1.1 be used instead of HTTP/2.

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

8. HTTP Message Exchanges

HTTP/2 is intended to be as compatible as possible with current uses of HTTP. This means that, from the application perspective, the features of the protocol are largely unchanged. To achieve this, all request and response semantics are preserved, although the syntax of conveying those semantics has changed.

Thus, the specification and requirements of HTTP/1.1 Semantics and Content [RFC7231], Conditional Requests [RFC7232], Range Requests [RFC7233], Caching [RFC7234] and Authentication [RFC7235] are applicable to HTTP/2. Selected portions of HTTP/1.1 Message Syntax and Routing [RFC7230], such as the HTTP and HTTPS URI schemes, are also applicable in HTTP/2, but the expression of those semantics for this protocol are defined in the sections below.

8.1. HTTP Request/Response Exchange

A client sends an HTTP request on a new stream, using a previously unused stream identifier (Section 5.1.1). A server sends an HTTP response on the same stream as the request.

An HTTP message (request or response) consists of:

1. for a response only, zero or more HEADERS frames (each followed by zero or more CONTINUATION frames) containing the message headers of informational (1xx) HTTP responses (see [RFC7230], Section 3.2 and [RFC7231], Section 6.2), and
2. one HEADERS frame (followed by zero or more CONTINUATION frames) containing the message headers (see [RFC7230], Section 3.2), and
3. zero or more DATA frames containing the payload body (see [RFC7230], Section 3.3), and
4. optionally, one HEADERS frame, followed by zero or more CONTINUATION frames containing the trailer-part, if present (see [RFC7230], Section 4.1.2).

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

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

HTTP/2 uses DATA frames to carry message payloads. The "chunked" transfer encoding defined in Section 4.1 of [RFC7230] MUST NOT be used in HTTP/2.

Trailing header fields are carried in a header block that also terminates the stream. Such a header block is a sequence starting with a HEADERS frame, followed by zero or more CONTINUATION frames, where the HEADERS frame bears an END_STREAM flag. Header blocks after the first that do not terminate the stream are not part of an HTTP request or response.

A HEADERS frame (and associated CONTINUATION frames) can only appear at the start or end of a stream. An endpoint that receives a HEADERS frame without the END_STREAM flag set after receiving a final (non-informational) status code MUST treat the corresponding request or response as malformed (Section 8.1.2.6).

An HTTP request/response exchange fully consumes a single stream. A request starts with the HEADERS frame that puts the stream into an "open" state. The request ends with a frame bearing END_STREAM, which causes the stream to become "half closed (local)" for the client and "half closed (remote)" for the server. A response starts

with a HEADERS frame and ends with a frame bearing END_STREAM, which places the stream in the "closed" state.

An HTTP response is complete after the server sends - or the client receives - a frame with the END_STREAM flag set (including any CONTINUATION frames needed to complete a header block). A server can send a complete response prior to the client sending an entire request if the response does not depend on any portion of the request that has not been sent and received. When this is true, a server MAY request that the client abort transmission of a request without error by sending a RST_STREAM with an error code of NO_ERROR after sending a complete response (i.e., a frame with the END_STREAM flag). Clients MUST NOT discard responses as a result of receiving such a RST_STREAM, though clients can always discard responses at their discretion for other reasons.

8.1.1.1. Upgrading From HTTP/2

HTTP/2 removes support for the 101 (Switching Protocols) informational status code ([RFC7231], Section 6.2.2).

The semantics of 101 (Switching Protocols) aren't applicable to a multiplexed protocol. Alternative protocols are able to use the same mechanisms that HTTP/2 uses to negotiate their use (see Section 3).

8.1.1.2. HTTP Header Fields

HTTP header fields carry information as a series of key-value pairs. For a listing of registered HTTP headers, see the Message Header Field Registry maintained at [4].

Just as in HTTP/1.x, header field names are strings of ASCII characters that are compared in a case-insensitive fashion. However, header field names MUST be converted to lowercase prior to their encoding in HTTP/2. A request or response containing uppercase header field names MUST be treated as malformed (Section 8.1.2.6).

8.1.2.1. Pseudo-Header Fields

While HTTP/1.x used the message start-line (see [RFC7230], Section 3.1) to convey the target URI and method of the request, and the status code for the response, HTTP/2 uses special pseudo-header fields beginning with ':' character (ASCII 0x3a) for this purpose.

Pseudo-header fields are not HTTP header fields. Endpoints MUST NOT generate pseudo-header fields other than those defined in this document.

Pseudo-header fields are only valid in the context in which they are defined. Pseudo-header fields defined for requests MUST NOT appear in responses; pseudo-header fields defined for responses MUST NOT appear in requests. Pseudo-header fields MUST NOT appear in trailers. Endpoints MUST treat a request or response that contains undefined or invalid pseudo-header fields as malformed (Section 8.1.2.6).

All pseudo-header fields MUST appear in the header block before regular header fields. Any request or response that contains a pseudo-header field that appears in a header block after a regular header field MUST be treated as malformed (Section 8.1.2.6).

8.1.2.2. Connection-Specific Header Fields

HTTP/2 does not use the "Connection" header field to indicate connection-specific header fields; in this protocol, connection-specific metadata is conveyed by other means. An endpoint MUST NOT generate an HTTP/2 message containing connection-specific header fields; any message containing connection-specific header fields MUST be treated as malformed (Section 8.1.2.6).

The only exception to this is the TE header field, which MAY be present in an HTTP/2 request; when it is, it MUST NOT contain any value other than "trailers".

This means that an intermediary transforming an HTTP/1.x message to HTTP/2 will need to remove any header fields nominated by the Connection header field, along with the Connection header field itself. Such intermediaries SHOULD also remove other connection-specific header fields, such as Keep-Alive, Proxy-Connection, Transfer-Encoding and Upgrade, even if they are not nominated by Connection.

Note: HTTP/2 purposefully does not support upgrade to another protocol. The handshake methods described in Section 3 are believed sufficient to negotiate the use of alternative protocols.

8.1.2.3. Request Pseudo-Header Fields

The following pseudo-header fields are defined for HTTP/2 requests:

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

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

- o The ":authority" pseudo-header field includes the authority portion of the target URI ([RFC3986], Section 3.2). The authority MUST NOT include the deprecated "userinfo" subcomponent for "http" or "https" schemes URIs.

To ensure that the HTTP/1.1 request line can be reproduced accurately, this pseudo-header field MUST be omitted when translating from an HTTP/1.1 request that has a request target in origin or asterisk form (see [RFC7230], Section 5.3). Clients that generate HTTP/2 requests directly SHOULD use the ":authority" pseudo-header field instead of the "Host" header field. An intermediary that converts an HTTP/2 request to HTTP/1.1 MUST create a "Host" header field if one is not present in a request by copying the value of the ":authority" pseudo-header field.

- o The ":path" pseudo-header field includes the path and query parts of the target URI (the "path-absolute" production from [RFC3986] and optionally a '?' character followed by the "query" production, see [RFC3986], Section 3.3 and [RFC3986], Section 3.4). A request in asterisk form includes the value '*' for the ":path" pseudo-header field.

This pseudo-header field MUST NOT be empty for "http" or "https" URIs; "http" or "https" URIs that do not contain a path component MUST include a value of '/'. The exception to this rule is an OPTIONS request for an "http" or "https" URI that does not include a path component; these MUST include a ":path" pseudo-header field with a value of '*' (see [RFC7230], Section 5.3.4).

All HTTP/2 requests MUST include exactly one valid value for the ":method", ":scheme", and ":path" pseudo-header fields, unless it is a CONNECT request (Section 8.3). An HTTP request that omits mandatory pseudo-header fields is malformed (Section 8.1.2.6).

HTTP/2 does not define a way to carry the version identifier that is included in the HTTP/1.1 request line.

8.1.2.4. Response Pseudo-Header Fields

For HTTP/2 responses, a single ":status" pseudo-header field is defined that carries the HTTP status code field (see [RFC7231], Section 6). This pseudo-header field MUST be included in all responses, otherwise the response is malformed (Section 8.1.2.6).

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

8.1.2.5. Compressing the Cookie Header Field

The Cookie header field [COOKIE] uses a semi-colon (";") to delimit cookie-pairs (or "crumbs"). This header field doesn't follow the list construction rules in HTTP (see [RFC7230], Section 3.2.2), which prevents cookie-pairs from being separated into different name-value pairs. This can significantly reduce compression efficiency as individual cookie-pairs are updated.

To allow for better compression efficiency, the Cookie header field MAY be split into separate header fields, each with one or more cookie-pairs. If there are multiple Cookie header fields after decompression, these MUST be concatenated into a single octet string using the two octet delimiter of 0x3B, 0x20 (the ASCII string "; ") before being passed into a non-HTTP/2 context, such as an HTTP/1.1 connection, or a generic HTTP server application.

Therefore, the following two lists of Cookie header fields are semantically equivalent.

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

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

8.1.2.6. Malformed Requests and Responses

A malformed request or response is one that is an otherwise valid sequence of HTTP/2 frames, but is otherwise invalid due to the presence of extraneous frames, prohibited header fields, the absence of mandatory header fields, or the inclusion of uppercase header field names.

A request or response that includes a payload body can include a "content-length" header field. A request or response is also malformed if the value of a "content-length" header field does not equal the sum of the DATA frame payload lengths that form the body. A response that is defined to have no payload, as described in [RFC7230], Section 3.3.2, can have a non-zero "content-length" header field, even though no content is included in DATA frames.

Intermediaries that process HTTP requests or responses (i.e., any intermediary not acting as a tunnel) MUST NOT forward a malformed request or response. Malformed requests or responses that are

detected MUST be treated as a stream error (Section 5.4.2) of type `PROTOCOL_ERROR`.

For malformed requests, a server MAY send an HTTP response prior to closing or resetting the stream. Clients MUST NOT accept a malformed response. Note that these requirements are intended to protect against several types of common attacks against HTTP; they are deliberately strict, because being permissive can expose implementations to these vulnerabilities.

8.1.3. Examples

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

An HTTP GET request includes request header fields and no payload body and is therefore transmitted as a single HEADERS frame, followed by zero or more CONTINUATION frames containing the serialized block of request header fields. The HEADERS frame in the following has both the `END_HEADERS` and `END_STREAM` flags set; no CONTINUATION frames are sent:

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

Similarly, a response that includes only response header fields is transmitted as a HEADERS frame (again, followed by zero or more CONTINUATION frames) containing the serialized block of response header fields.

```
HTTP/1.1 304 Not Modified       HEADERS
ETag: "xyzzy"                  ==>  + END_STREAM
Expires: Thu, 23 Jan ...       + END_HEADERS
                                :status = 304
                                etag = "xyzzy"
                                expires = Thu, 23 Jan ...
```

An HTTP POST request that includes request header fields and payload data is transmitted as one HEADERS frame, followed by zero or more CONTINUATION frames containing the request header fields, followed by one or more DATA frames, with the last CONTINUATION (or HEADERS)

frame having the END_HEADERS flag set and the final DATA frame having the END_STREAM flag set:

```
POST /resource HTTP/1.1      HEADERS
Host: example.org            ==>  - END_STREAM
Content-Type: image/jpeg      - END_HEADERS
Content-Length: 123           :method = POST
                               :path = /resource
                               :scheme = https
{binary data}

CONTINUATION
+ END_HEADERS
  content-type = image/jpeg
  host = example.org
  content-length = 123

DATA
+ END_STREAM
{binary data}
```

Note that data contributing to any given header field could be spread between header block fragments. The allocation of header fields to frames in this example is illustrative only.

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

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

DATA
+ END_STREAM
{binary data}
```

An informational response using a 1xx status code other than 101 is transmitted as a HEADERS frame, followed by zero or more CONTINUATION frames.

Trailing header fields are sent as a header block after both the request or response header block and all the DATA frames have been sent. The HEADERS frame starting the trailers header block has the END_STREAM flag set.

The following example includes both a 100 (Continue) status code, which is sent in response to a request containing a "100-continue" token in the Expect header field, and trailing header fields:

HTTP/1.1 100 Continue		HEADERS
Extension-Field: bar	==>	- END_STREAM
		+ END_HEADERS
		:status = 100
		extension-field = bar
HTTP/1.1 200 OK		HEADERS
Content-Type: image/jpeg	==>	- END_STREAM
Transfer-Encoding: chunked		+ END_HEADERS
Trailer: Foo		:status = 200
		content-length = 123
123		content-type = image/jpeg
{binary data}		trailer = Foo
0		
Foo: bar		DATA
		- END_STREAM
		{binary data}
		HEADERS
		+ END_STREAM
		+ END_HEADERS
		foo = bar

8.1.4. Request Reliability Mechanisms in HTTP/2

In HTTP/1.1, an HTTP client is unable to retry a non-idempotent request when an error occurs, because there is no means to determine the nature of the error. It is possible that some server processing occurred prior to the error, which could result in undesirable effects if the request were reattempted.

HTTP/2 provides two mechanisms for providing a guarantee to a client that a request has not been processed:

- o The GOAWAY frame indicates the highest stream number that might have been processed. Requests on streams with higher numbers are therefore guaranteed to be safe to retry.
- o The REFUSED_STREAM error code can be included in a RST_STREAM frame to indicate that the stream is being closed prior to any processing having occurred. Any request that was sent on the reset stream can be safely retried.

Requests that have not been processed have not failed; clients MAY automatically retry them, even those with non-idempotent methods.

A server MUST NOT indicate that a stream has not been processed unless it can guarantee that fact. If frames that are on a stream are passed to the application layer for any stream, then `REFUSED_STREAM` MUST NOT be used for that stream, and a `GOAWAY` frame MUST include a stream identifier that is greater than or equal to the given stream identifier.

In addition to these mechanisms, the `PING` frame provides a way for a client to easily test a connection. Connections that remain idle can become broken as some middleboxes (for instance, network address translators, or load balancers) silently discard connection bindings. The `PING` frame allows a client to safely test whether a connection is still active without sending a request.

8.2. Server Push

HTTP/2 allows a server to pre-emptively send (or "push") responses (along with corresponding "promised" requests) to a client in association with a previous client-initiated request. This can be useful when the server knows the client will need to have those responses available in order to fully process the response to the original request.

A client can request that server push be disabled, though this is negotiated for each hop independently. The `SETTINGS_ENABLE_PUSH` setting can be set to 0 to indicate that server push is disabled.

Promised requests MUST be cacheable (see [RFC7231], Section 4.2.3), MUST be safe (see [RFC7231], Section 4.2.1) and MUST NOT include a request body. Clients that receive a promised request that is not cacheable, is not known to be safe or that indicates the presence of a request body MUST reset the promised stream with a stream error (Section 5.4.2) of type `PROTOCOL_ERROR`. Note this could result in the promised stream being reset if the client does not recognize a newly defined method as being safe.

Pushed responses that are cacheable (see [RFC7234], Section 3) can be stored by the client, if it implements an HTTP cache. Pushed responses are considered successfully validated on the origin server (e.g., if the "no-cache" cache response directive [RFC7234], Section 5.2.2 is present) while the stream identified by the promised stream ID is still open.

Pushed responses that are not cacheable MUST NOT be stored by any HTTP cache. They MAY be made available to the application separately.

The server MUST include a value in the ":authority" header field for which the server is authoritative (see Section 10.1). A client MUST treat a PUSH_PROMISE for which the server is not authoritative as a stream error (Section 5.4.2) of type `PROTOCOL_ERROR`.

An intermediary can receive pushes from the server and choose not to forward them on to the client. In other words, how to make use of the pushed information is up to that intermediary. Equally, the intermediary might choose to make additional pushes to the client, without any action taken by the server.

A client cannot push. Thus, servers MUST treat the receipt of a PUSH_PROMISE frame as a connection error (Section 5.4.1) of type `PROTOCOL_ERROR`. Clients MUST reject any attempt to change the `SETTINGS_ENABLE_PUSH` setting to a value other than 0 by treating the message as a connection error (Section 5.4.1) of type `PROTOCOL_ERROR`.

8.2.1. Push Requests

Server push is semantically equivalent to a server responding to a request; however, in this case that request is also sent by the server, as a PUSH_PROMISE frame.

The PUSH_PROMISE frame includes a header block that contains a complete set of request header fields that the server attributes to the request. It is not possible to push a response to a request that includes a request body.

Pushed responses are always associated with an explicit request from the client. The PUSH_PROMISE frames sent by the server are sent on that explicit request's stream. The PUSH_PROMISE frame also includes a promised stream identifier, chosen from the stream identifiers available to the server (see Section 5.1.1).

The header fields in PUSH_PROMISE and any subsequent CONTINUATION frames MUST be a valid and complete set of request header fields (Section 8.1.2.3). The server MUST include a method in the ":method" header field that is safe and cacheable. If a client receives a PUSH_PROMISE that does not include a complete and valid set of header fields, or the ":method" header field identifies a method that is not safe, it MUST respond with a stream error (Section 5.4.2) of type `PROTOCOL_ERROR`.

The server SHOULD send PUSH_PROMISE (Section 6.6) frames prior to sending any frames that reference the promised responses. This avoids a race where clients issue requests prior to receiving any PUSH_PROMISE frames.

For example, if the server receives a request for a document containing embedded links to multiple image files, and the server chooses to push those additional images to the client, sending push promises before the DATA frames that contain the image links ensures that the client is able to see the promises before discovering embedded links. Similarly, if the server pushes responses referenced by the header block (for instance, in Link header fields), sending the push promises before sending the header block ensures that clients do not request them.

PUSH_PROMISE frames MUST NOT be sent by the client.

PUSH_PROMISE frames can be sent by the server in response to any client-initiated stream, but the stream MUST be in either the "open" or "half closed (remote)" state with respect to the server. PUSH_PROMISE frames are interspersed with the frames that comprise a response, though they cannot be interspersed with HEADERS and CONTINUATION frames that comprise a single header block.

Sending a PUSH_PROMISE frame creates a new stream and puts the stream into the "reserved (local)" state for the server and the "reserved (remote)" state for the client.

8.2.2. Push Responses

After sending the PUSH_PROMISE frame, the server can begin delivering the pushed response as a response (Section 8.1.2.4) on a server-initiated stream that uses the promised stream identifier. The server uses this stream to transmit an HTTP response, using the same sequence of frames as defined in Section 8.1. This stream becomes "half closed" to the client (Section 5.1) after the initial HEADERS frame is sent.

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

If the client determines, for any reason, that it does not wish to receive the pushed response from the server, or if the server takes too long to begin sending the promised response, the client can send an RST_STREAM frame, using either the CANCEL or REFUSED_STREAM codes, and referencing the pushed stream's identifier.

A client can use the `SETTINGS_MAX_CONCURRENT_STREAMS` setting to limit the number of responses that can be concurrently pushed by a server. Advertising a `SETTINGS_MAX_CONCURRENT_STREAMS` value of zero disables server push by preventing the server from creating the necessary streams. This does not prohibit a server from sending `PUSH_PROMISE` frames; clients need to reset any promised streams that are not wanted.

Clients receiving a pushed response MUST validate that either the server is authoritative (see Section 10.1), or the proxy that provided the pushed response is configured for the corresponding request. For example, a server that offers a certificate for only the "example.com" DNS-ID or Common Name is not permitted to push a response for "https://www.example.org/doc".

The response for a `PUSH_PROMISE` stream begins with a `HEADERS` frame, which immediately puts the stream into the "half closed (remote)" state for the server and "half closed (local)" state for the client, and ends with a frame bearing `END_STREAM`, which places the stream in the "closed" state.

Note: The client never sends a frame with the `END_STREAM` flag for a server push.

8.3. The CONNECT Method

In HTTP/1.x, the pseudo-method `CONNECT` ([RFC7231], Section 4.3.6) is used to convert an HTTP connection into a tunnel to a remote host. `CONNECT` is primarily used with HTTP proxies to establish a TLS session with an origin server for the purposes of interacting with "https" resources.

In HTTP/2, the `CONNECT` method is used to establish a tunnel over a single HTTP/2 stream to a remote host, for similar purposes. The HTTP header field mapping works as defined in Request Header Fields (Section 8.1.2.3), with a few differences. Specifically:

- o The `":method"` header field is set to `"CONNECT"`.
- o The `":scheme"` and `":path"` header fields MUST be omitted.
- o The `":authority"` header field contains the host and port to connect to (equivalent to the authority-form of the request-target of `CONNECT` requests, see [RFC7230], Section 5.3).

A `CONNECT` request that does not conform to these restrictions is malformed (Section 8.1.2.6).

A proxy that supports CONNECT establishes a TCP connection [TCP] to the server identified in the ":authority" header field. Once this connection is successfully established, the proxy sends a HEADERS frame containing a 2xx series status code to the client, as defined in [RFC7231], Section 4.3.6.

After the initial HEADERS frame sent by each peer, all subsequent DATA frames correspond to data sent on the TCP connection. The payload of any DATA frames sent by the client is transmitted by the proxy to the TCP server; data received from the TCP server is assembled into DATA frames by the proxy. Frame types other than DATA or stream management frames (RST_STREAM, WINDOW_UPDATE, and PRIORITY) MUST NOT be sent on a connected stream, and MUST be treated as a stream error (Section 5.4.2) if received.

The TCP connection can be closed by either peer. The END_STREAM flag on a DATA frame is treated as being equivalent to the TCP FIN bit. A client is expected to send a DATA frame with the END_STREAM flag set after receiving a frame bearing the END_STREAM flag. A proxy that receives a DATA frame with the END_STREAM flag set sends the attached data with the FIN bit set on the last TCP segment. A proxy that receives a TCP segment with the FIN bit set sends a DATA frame with the END_STREAM flag set. Note that the final TCP segment or DATA frame could be empty.

A TCP connection error is signaled with RST_STREAM. A proxy treats any error in the TCP connection, which includes receiving a TCP segment with the RST bit set, as a stream error (Section 5.4.2) of type CONNECT_ERROR. Correspondingly, a proxy MUST send a TCP segment with the RST bit set if it detects an error with the stream or the HTTP/2 connection.

9. Additional HTTP Requirements/Considerations

This section outlines attributes of the HTTP protocol that improve interoperability, reduce exposure to known security vulnerabilities, or reduce the potential for implementation variation.

9.1. Connection Management

HTTP/2 connections are persistent. For best performance, it is expected clients will not close connections until it is determined that no further communication with a server is necessary (for example, when a user navigates away from a particular web page), or until the server closes the connection.

Clients SHOULD NOT open more than one HTTP/2 connection to a given host and port pair, where host is derived from a URI, a selected alternative service [ALT-SVC], or a configured proxy.

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

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

Servers are encouraged to maintain open connections for as long as possible, but are permitted to terminate idle connections if necessary. When either endpoint chooses to close the transport-layer TCP connection, the terminating endpoint SHOULD first send a GOAWAY (Section 6.8) frame so that both endpoints can reliably determine whether previously sent frames have been processed and gracefully complete or terminate any necessary remaining tasks.

9.1.1.1. Connection Reuse

Connections that are made to an origin server, either directly or through a tunnel created using the CONNECT method (Section 8.3) MAY be reused for requests with multiple different URI authority components. A connection can be reused as long as the origin server is authoritative (Section 10.1). For TCP connections without TLS, this depends on the host having resolved to the same IP address.

For "https" resources, connection reuse additionally depends on having a certificate that is valid for the host in the URI. The certificate presented by the server MUST satisfy any checks that the client would perform when forming a new TLS connection for the host in the URI.

An origin server might offer a certificate with multiple "subjectAltName" attributes, or names with wildcards, one of which is valid for the authority in the URI. For example, a certificate with a "subjectAltName" of "*.example.com" might permit the use of the same connection for requests to URIs starting with "https://a.example.com/" and "https://b.example.com/".

In some deployments, reusing a connection for multiple origins can result in requests being directed to the wrong origin server. For example, TLS termination might be performed by a middlebox that uses

the TLS Server Name Indication (SNI) [TLS-EXT] extension to select an origin server. This means that it is possible for clients to send confidential information to servers that might not be the intended target for the request, even though the server is otherwise authoritative.

A server that does not wish clients to reuse connections can indicate that it is not authoritative for a request by sending a 421 (Misdirected Request) status code in response to the request (see Section 9.1.2).

A client that is configured to use a proxy over HTTP/2 directs requests to that proxy through a single connection. That is, all requests sent via a proxy reuse the connection to the proxy.

9.1.2. The 421 (Misdirected Request) Status Code

The 421 (Misdirected Request) status code indicates that the request was directed at a server that is not able to produce a response. This can be sent by a server that is not configured to produce responses for the combination of scheme and authority that are included in the request URI.

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

This status code MUST NOT be generated by proxies.

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

9.2. Use of TLS Features

Implementations of HTTP/2 MUST use TLS [TLS12] version 1.2 or higher for HTTP/2 over TLS. The general TLS usage guidance in [TLSBCP] SHOULD be followed, with some additional restrictions that are specific to HTTP/2.

The TLS implementation MUST support the Server Name Indication (SNI) [TLS-EXT] extension to TLS. HTTP/2 clients MUST indicate the target domain name when negotiating TLS.

Deployments of HTTP/2 that negotiate TLS 1.3 or higher need only support and use the SNI extension; deployments of TLS 1.2 are subject

to the requirements in the following sections. Implementations are encouraged to provide defaults that comply, but it is recognized that deployments are ultimately responsible for compliance.

9.2.1. TLS 1.2 Features

This section describes restrictions on the TLS 1.2 feature set that can be used with HTTP/2. Due to deployment limitations, it might not be possible to fail TLS negotiation when these restrictions are not met. An endpoint MAY immediately terminate an HTTP/2 connection that does not meet these TLS requirements with a connection error (Section 5.4.1) of type `INADEQUATE_SECURITY`.

A deployment of HTTP/2 over TLS 1.2 MUST disable compression. TLS compression can lead to the exposure of information that would not otherwise be revealed [RFC3749]. Generic compression is unnecessary since HTTP/2 provides compression features that are more aware of context and therefore likely to be more appropriate for use for performance, security or other reasons.

A deployment of HTTP/2 over TLS 1.2 MUST disable renegotiation. An endpoint MUST treat a TLS renegotiation as a connection error (Section 5.4.1) of type `PROTOCOL_ERROR`. Note that disabling renegotiation can result in long-lived connections becoming unusable due to limits on the number of messages the underlying cipher suite can encipher.

An endpoint MAY use renegotiation to provide confidentiality protection for client credentials offered in the handshake, but any renegotiation MUST occur prior to sending the connection preface. A server SHOULD request a client certificate if it sees a renegotiation request immediately after establishing a connection.

This effectively prevents the use of renegotiation in response to a request for a specific protected resource. A future specification might provide a way to support this use case. Alternatively, a server might use an error (Section 5.4) of type `HTTP_1_1_REQUIRED` to request the client use a protocol which supports renegotiation.

Implementations MUST support ephemeral key exchange sizes of at least 2048 bits for cipher suites that use ephemeral finite field Diffie-Hellman (DHE) [TLS12] and 224 bits for cipher suites that use ephemeral elliptic curve Diffie-Hellman (ECDHE) [RFC4492]. Clients MUST accept DHE sizes of up to 4096 bits. Endpoints MAY treat negotiation of key sizes smaller than the lower limits as a connection error (Section 5.4.1) of type `INADEQUATE_SECURITY`.

9.2.2. TLS 1.2 Cipher Suites

A deployment of HTTP/2 over TLS 1.2 SHOULD NOT use any of the cipher suites that are listed in the cipher suite black list (Appendix A).

Endpoints MAY choose to generate a connection error (Section 5.4.1) of type INADEQUATE_SECURITY if one of the cipher suites from the black list are negotiated. A deployment that chooses to use a black-listed cipher suite risks triggering a connection error unless the set of potential peers is known to accept that cipher suite.

Implementations MUST NOT generate this error in reaction to the negotiation of a cipher suite that is not on the black list. Consequently, when clients offer a cipher suite that is not on the black list, they have to be prepared to use that cipher suite with HTTP/2.

The black list includes the cipher suite that TLS 1.2 makes mandatory, which means that TLS 1.2 deployments could have non-intersecting sets of permitted cipher suites. To avoid this problem causing TLS handshake failures, deployments of HTTP/2 that use TLS 1.2 MUST support TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 [TLS-ECDHE] with the P256 elliptic curve [FIPS186].

Note that clients might advertise support of cipher suites that are on the black list in order to allow for connection to servers that do not support HTTP/2. This allows servers to select HTTP/1.1 with a cipher suite that is on the HTTP/2 black list. However, this can result in HTTP/2 being negotiated with a black-listed cipher suite if the application protocol and cipher suite are independently selected.

10. Security Considerations

10.1. Server Authority

HTTP/2 relies on the HTTP/1.1 definition of authority for determining whether a server is authoritative in providing a given response, see [RFC7230], Section 9.1. This relies on local name resolution for the "http" URI scheme, and the authenticated server identity for the "https" scheme (see [RFC2818], Section 3).

10.2. Cross-Protocol Attacks

In a cross-protocol attack, an attacker causes a client to initiate a transaction in one protocol toward a server that understands a different protocol. An attacker might be able to cause the transaction to appear as valid transaction in the second protocol.

In combination with the capabilities of the web context, this can be used to interact with poorly protected servers in private networks.

Completing a TLS handshake with an ALPN identifier for HTTP/2 can be considered sufficient protection against cross protocol attacks. ALPN provides a positive indication that a server is willing to proceed with HTTP/2, which prevents attacks on other TLS-based protocols.

The encryption in TLS makes it difficult for attackers to control the data which could be used in a cross-protocol attack on a cleartext protocol.

The cleartext version of HTTP/2 has minimal protection against cross-protocol attacks. The connection preface (Section 3.5) contains a string that is designed to confuse HTTP/1.1 servers, but no special protection is offered for other protocols. A server that is willing to ignore parts of an HTTP/1.1 request containing an Upgrade header field in addition to the client connection preface could be exposed to a cross-protocol attack.

10.3. Intermediary Encapsulation Attacks

The HTTP/2 header field encoding allows the expression of names that are not valid field names in the Internet Message Syntax used by HTTP/1.1. Requests or responses containing invalid header field names MUST be treated as malformed (Section 8.1.2.6). An intermediary therefore cannot translate an HTTP/2 request or response containing an invalid field name into an HTTP/1.1 message.

Similarly, HTTP/2 allows header field values that are not valid. While most of the values that can be encoded will not alter header field parsing, carriage return (CR, ASCII 0xd), line feed (LF, ASCII 0xa), and the zero character (NUL, ASCII 0x0) might be exploited by an attacker if they are translated verbatim. Any request or response that contains a character not permitted in a header field value MUST be treated as malformed (Section 8.1.2.6). Valid characters are defined by the "field-content" ABNF rule in Section 3.2 of [RFC7230].

10.4. Cacheability of Pushed Responses

Pushed responses do not have an explicit request from the client; the request is provided by the server in the PUSH_PROMISE frame.

Caching responses that are pushed is possible based on the guidance provided by the origin server in the Cache-Control header field. However, this can cause issues if a single server hosts more than one

tenant. For example, a server might offer multiple users each a small portion of its URI space.

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

Pushed responses for which an origin server is not authoritative (see Section 10.1) MUST NOT be used or cached.

10.5. Denial of Service Considerations

An HTTP/2 connection can demand a greater commitment of resources to operate than a HTTP/1.1 connection. The use of header compression and flow control depend on a commitment of resources for storing a greater amount of state. Settings for these features ensure that memory commitments for these features are strictly bounded.

The number of PUSH_PROMISE frames is not constrained in the same fashion. A client that accepts server push SHOULD limit the number of streams it allows to be in the "reserved (remote)" state. Excessive number of server push streams can be treated as a stream error (Section 5.4.2) of type ENHANCE_YOUR_CALM.

Processing capacity cannot be guarded as effectively as state capacity.

The SETTINGS frame can be abused to cause a peer to expend additional processing time. This might be done by pointlessly changing SETTINGS parameters, setting multiple undefined parameters, or changing the same setting multiple times in the same frame. WINDOW_UPDATE or PRIORITY frames can be abused to cause an unnecessary waste of resources.

Large numbers of small or empty frames can be abused to cause a peer to expend time processing frame headers. Note however that some uses are entirely legitimate, such as the sending of an empty DATA or CONTINUATION frame at the end of a stream.

Header compression also offers some opportunities to waste processing resources; see Section 7 of [COMPRESSION] for more details on potential abuses.

Limits in SETTINGS parameters cannot be reduced instantaneously, which leaves an endpoint exposed to behavior from a peer that could

exceed the new limits. In particular, immediately after establishing a connection, limits set by a server are not known to clients and could be exceeded without being an obvious protocol violation.

All these features - i.e., SETTINGS changes, small frames, header compression - have legitimate uses. These features become a burden only when they are used unnecessarily or to excess.

An endpoint that doesn't monitor this behavior exposes itself to a risk of denial of service attack. Implementations SHOULD track the use of these features and set limits on their use. An endpoint MAY treat activity that is suspicious as a connection error (Section 5.4.1) of type `ENHANCE_YOUR_CALM`.

10.5.1. Limits on Header Block Size

A large header block (Section 4.3) can cause an implementation to commit a large amount of state. Header fields that are critical for routing can appear toward the end of a header block, which prevents streaming of header fields to their ultimate destination. This ordering and other reasons, such as ensuring cache correctness, means that an endpoint might need to buffer the entire header block. Since there is no hard limit to the size of a header block, some endpoints could be forced to commit a large amount of available memory for header fields.

An endpoint can use the `SETTINGS_MAX_HEADER_LIST_SIZE` to advise peers of limits that might apply on the size of header blocks. This setting is only advisory, so endpoints MAY choose to send header blocks that exceed this limit and risk having the request or response being treated as malformed. This setting is specific to a connection, so any request or response could encounter a hop with a lower, unknown limit. An intermediary can attempt to avoid this problem by passing on values presented by different peers, but they are not obligated to do so.

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

10.5.2. CONNECT Issues

The CONNECT method can be used to create disproportionate load on an proxy, since stream creation is relatively inexpensive when compared to the creation and maintenance of a TCP connection. A proxy might also maintain some resources for a TCP connection beyond the closing

of the stream that carries the CONNECT request, since the outgoing TCP connection remains in the TIME_WAIT state. A proxy therefore cannot rely on SETTINGS_MAX_CONCURRENT_STREAMS alone to limit the resources consumed by CONNECT requests.

10.6. Use of Compression

Compression can allow an attacker to recover secret data when it is compressed in the same context as data under attacker control. HTTP/2 enables compression of header fields (Section 4.3); the following concerns also apply to the use of HTTP compressed content-codings ([RFC7231], Section 3.1.2.1).

There are demonstrable attacks on compression that exploit the characteristics of the web (e.g., [BREACH]). The attacker induces multiple requests containing varying plaintext, observing the length of the resulting ciphertext in each, which reveals a shorter length when a guess about the secret is correct.

Implementations communicating on a secure channel MUST NOT compress content that includes both confidential and attacker-controlled data unless separate compression dictionaries are used for each source of data. Compression MUST NOT be used if the source of data cannot be reliably determined. Generic stream compression, such as that provided by TLS MUST NOT be used with HTTP/2 (see Section 9.2).

Further considerations regarding the compression of header fields are described in [COMPRESSION].

10.7. Use of Padding

Padding within HTTP/2 is not intended as a replacement for general purpose padding, such as might be provided by TLS [TLS12]. Redundant padding could even be counterproductive. Correct application can depend on having specific knowledge of the data that is being padded.

To mitigate attacks that rely on compression, disabling or limiting compression might be preferable to padding as a countermeasure.

Padding can be used to obscure the exact size of frame content, and is provided to mitigate specific attacks within HTTP. For example, attacks where compressed content includes both attacker-controlled plaintext and secret data (see for example, [BREACH]).

Use of padding can result in less protection than might seem immediately obvious. At best, padding only makes it more difficult for an attacker to infer length information by increasing the number of frames an attacker has to observe. Incorrectly implemented

padding schemes can be easily defeated. In particular, randomized padding with a predictable distribution provides very little protection; similarly, padding payloads to a fixed size exposes information as payload sizes cross the fixed size boundary, which could be possible if an attacker can control plaintext.

Intermediaries SHOULD retain padding for DATA frames, but MAY drop padding for HEADERS and PUSH_PROMISE frames. A valid reason for an intermediary to change the amount of padding of frames is to improve the protections that padding provides.

10.8. Privacy Considerations

Several characteristics of HTTP/2 provide an observer an opportunity to correlate actions of a single client or server over time. This includes the value of settings, the manner in which flow control windows are managed, the way priorities are allocated to streams, timing of reactions to stimulus, and handling of any features that are controlled by settings.

As far as this creates observable differences in behavior, they could be used as a basis for fingerprinting a specific client, as defined in Section 1.8 of [HTML5].

HTTP/2's preference for using a single TCP connection allows correlation of a user's activity on a site. If connections are reused for different origins, this allows tracking across those origins.

Because the PING and SETTINGS frames solicit immediate responses, they can be used by an endpoint to measure latency to their peer. This might have privacy implications in certain scenarios.

11. IANA Considerations

A string for identifying HTTP/2 is entered into the "Application Layer Protocol Negotiation (ALPN) Protocol IDs" registry established in [TLS-ALPN].

This document establishes a registry for frame types, settings, and error codes. These new registries are entered into a new "Hypertext Transfer Protocol (HTTP) 2 Parameters" section.

This document registers the "HTTP2-Settings" header field for use in HTTP; and the 421 (Misdirected Request) status code.

This document registers the "PRI" method for use in HTTP, to avoid collisions with the connection preface (Section 3.5).

11.1. Registration of HTTP/2 Identification Strings

This document creates two registrations for the identification of HTTP/2 in the "Application Layer Protocol Negotiation (ALPN) Protocol IDs" registry established in [TLS-ALPN].

The "h2" string identifies HTTP/2 when used over TLS:

Protocol: HTTP/2 over TLS

Identification Sequence: 0x68 0x32 ("h2")

Specification: This document

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

Protocol: HTTP/2 over TCP

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

Specification: This document

11.2. Frame Type Registry

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

New entries in this registry require the following information:

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

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

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

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

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

11.3. Settings Registry

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

New registrations are advised to provide the following information:

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

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

Initial Value: An initial value for the setting.

Specification: An optional reference to a specification that describes the use of the setting.

An initial set of setting registrations can be found in Section 6.5.2.

Name	Code	Initial Value	Specification
HEADER_TABLE_SIZE	0x1	4096	Section 6.5.2
ENABLE_PUSH	0x2	1	Section 6.5.2
MAX_CONCURRENT_STREAMS	0x3	(infinite)	Section 6.5.2
INITIAL_WINDOW_SIZE	0x4	65535	Section 6.5.2
MAX_FRAME_SIZE	0x5	16384	Section 6.5.2
MAX_HEADER_LIST_SIZE	0x6	(infinite)	Section 6.5.2

11.4. Error Code Registry

This document establishes a registry for HTTP/2 error codes. The "HTTP/2 Error Code" registry manages a 32-bit space. The "HTTP/2 Error Code" registry operates under the "Expert Review" policy [RFC5226].

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

New registrations are advised to provide the following information:

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

Code: The 32-bit error code value.

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

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

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

Name	Code	Description	Specification
NO_ERROR	0x0	Graceful shutdown	Section 7
PROTOCOL_ERROR	0x1	Protocol error detected	Section 7
INTERNAL_ERROR	0x2	Implementation fault	Section 7
FLOW_CONTROL_ERROR	0x3	Flow control limits exceeded	Section 7
SETTINGS_TIMEOUT	0x4	Settings not acknowledged	Section 7
STREAM_CLOSED	0x5	Frame received for closed stream	Section 7
FRAME_SIZE_ERROR	0x6	Frame size incorrect	Section 7
REFUSED_STREAM	0x7	Stream not processed	Section 7
CANCEL	0x8	Stream cancelled	Section 7
COMPRESSION_ERROR	0x9	Compression state not updated	Section 7
CONNECT_ERROR	0xa	TCP connection error for CONNECT method	Section 7
ENHANCE_YOUR_CALM	0xb	Processing capacity exceeded	Section 7
INADEQUATE_SECURITY	0xc	Negotiated TLS parameters not acceptable	Section 7
HTTP_1_1_REQUIRED	0xd	Use HTTP/1.1 for the request	Section 7

11.5. HTTP2-Settings Header Field Registration

This section registers the "HTTP2-Settings" header field in the Permanent Message Header Field Registry [BCP90].

Header field name: HTTP2-Settings

Applicable protocol: http

Status: standard

Author/Change controller: IETF

Specification document(s): Section 3.2.1 of this document

Related information: This header field is only used by an HTTP/2 client for Upgrade-based negotiation.

11.6. PRI Method Registration

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

Method Name: PRI

Safe Yes

Idempotent Yes

Specification document(s) Section 3.5 of this document

Related information: This method is never used by an actual client. This method will appear to be used when an HTTP/1.1 server or intermediary attempts to parse an HTTP/2 connection preface.

11.7. The 421 (Misdirected Request) HTTP Status Code

This document registers the 421 (Misdirected Request) HTTP Status code in the Hypertext Transfer Protocol (HTTP) Status Code Registry ([RFC7231], Section 8.2).

Status Code: 421

Short Description: Misdirected Request

Specification: Section 9.1.2 of this document

12. Acknowledgements

This document includes substantial input from the following individuals:

- o Adam Langley, Wan-Teh Chang, Jim Morrison, Mark Nottingham, Alyssa Wilk, Costin Manolache, William Chan, Vitaliy Lvin, Joe Chan, Adam Barth, Ryan Hamilton, Gavin Peters, Kent Alstad, Kevin Lindsay, Paul Amer, Fan Yang, Jonathan Leighton (SPDY contributors).
- o Gabriel Montenegro and Willy Tarreau (Upgrade mechanism).
- o William Chan, Salvatore Loreto, Osama Mazahir, Gabriel Montenegro, Jitu Padhye, Roberto Peon, Rob Trace (Flow control).
- o Mike Bishop (Extensibility).
- o Mark Nottingham, Julian Reschke, James Snell, Jeff Pinner, Mike Bishop, Herve Ruellan (Substantial editorial contributions).

- o Kari Hurtta, Tatsuhiro Tsujikawa, Greg Wilkins, Poul-Henning Kamp, Jonathan Thackray.
- o Alexey Melnikov was an editor of this document during 2013.
- o A substantial proportion of Martin's contribution was supported by Microsoft during his employment there.
- o The Japanese HTTP/2 community provided an invaluable contribution, including a number of implementations, plus numerous technical and editorial contributions.

13. References

13.1. Normative References

[COMPRESSION]

Ruellan, H. and R. Peon, "HPACK - Header Compression for HTTP/2", draft-ietf-httpbis-header-compression-11 (work in progress), February 2015.

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

[FIPS186] NIST, "Digital Signature Standard (DSS)", FIPS PUB 186-4, July 2013, <<http://dx.doi.org/10.6028/NIST.FIPS.186-4>>.

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

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.

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

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.

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

[RFC5234] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008.

- [RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, June 2014.
- [RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, June 2014.
- [RFC7232] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests", RFC 7232, June 2014.
- [RFC7233] Fielding, R., Ed., Lafon, Y., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Range Requests", RFC 7233, June 2014.
- [RFC7234] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Caching", RFC 7234, June 2014.
- [RFC7235] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Authentication", RFC 7235, June 2014.
- [TCP] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.
- [TLS-ALPN] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, July 2014.
- [TLS-ECDHE] Rescorla, E., "TLS Elliptic Curve Cipher Suites with SHA-256/384 and AES Galois Counter Mode (GCM)", RFC 5289, August 2008.
- [TLS-EXT] Eastlake, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, January 2011.
- [TLS12] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

13.2. Informative References

- [ALT-SVC] Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", draft-ietf-httpbis-alt-svc-06 (work in progress), February 2015.

- [BCP90] Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, September 2004.
- [BREACH] Gluck, Y., Harris, N., and A. Prado, "BREACH: Reviving the CRIME Attack", July 2013, <<http://breachattack.com/resources/BREACH%20-%20SSL,%20gone%20in%2030%20seconds.pdf>>.
- [HTML5] Hickson, I., Berjon, R., Faulkner, S., Leithead, T., Doyle Navara, E., O'Connor, E., and S. Pfeiffer, "HTML5", W3C Recommendation REC-html5-20141028, October 2014, <<http://www.w3.org/TR/2014/REC-html5-20141028/>>.
- Latest version available at [5].
- [RFC3749] Hollenbeck, S., "Transport Layer Security Protocol Compression Methods", RFC 3749, May 2004.
- [RFC4492] Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., and B. Moeller, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)", RFC 4492, May 2006.
- [RFC6585] Nottingham, M. and R. Fielding, "Additional HTTP Status Codes", RFC 6585, April 2012.
- [RFC7323] Borman, D., Braden, B., Jacobson, V., and R. Scheffenegger, "TCP Extensions for High Performance", RFC 7323, September 2014.
- [TALKING] Huang, L-S., Chen, E., Barth, A., Rescorla, E., and C. Jackson, "Talking to Yourself for Fun and Profit", 2011, <<http://w2spconf.com/2011/papers/websocket.pdf>>.
- [TLSBCP] Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of TLS and DTLS", draft-ietf-uta-tls-bcp-08 (work in progress), December 2014.

13.3. URIs

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

Appendix A. TLS 1.2 Cipher Suite Black List

An HTTP/2 implementation MAY treat the negotiation of any of the following cipher suites with TLS 1.2 as a connection error (Section 5.4.1) of type INADEQUATE_SECURITY: TLS_NULL_WITH_NULL_NULL, TLS_RSA_WITH_NULL_MD5, TLS_RSA_WITH_NULL_SHA, TLS_RSA_EXPORT_WITH_RC4_40_MD5, TLS_RSA_WITH_RC4_128_MD5, TLS_RSA_WITH_RC4_128_SHA, TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5, TLS_RSA_WITH_IDEA_CBC_SHA, TLS_RSA_EXPORT_WITH_DES40_CBC_SHA, TLS_RSA_WITH_DES_CBC_SHA, TLS_RSA_WITH_3DES_EDE_CBC_SHA, TLS_DH_DSS_EXPORT_WITH_DES40_CBC_SHA, TLS_DH_DSS_WITH_DES_CBC_SHA, TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA, TLS_DH_RSA_EXPORT_WITH_DES40_CBC_SHA, TLS_DH_RSA_WITH_DES_CBC_SHA, TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA, TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA, TLS_DHE_DSS_WITH_DES_CBC_SHA, TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA, TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA, TLS_DHE_RSA_WITH_DES_CBC_SHA, TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA, TLS_DH_anon_EXPORT_WITH_RC4_40_MD5, TLS_DH_anon_WITH_RC4_128_MD5, TLS_DH_anon_EXPORT_WITH_DES40_CBC_SHA, TLS_DH_anon_WITH_DES_CBC_SHA, TLS_DH_anon_WITH_3DES_EDE_CBC_SHA, TLS_KRB5_WITH_DES_CBC_SHA, TLS_KRB5_WITH_3DES_EDE_CBC_SHA, TLS_KRB5_WITH_RC4_128_SHA, TLS_KRB5_WITH_IDEA_CBC_SHA, TLS_KRB5_WITH_DES_CBC_MD5, TLS_KRB5_WITH_3DES_EDE_CBC_MD5, TLS_KRB5_WITH_RC4_128_MD5, TLS_KRB5_WITH_IDEA_CBC_MD5, TLS_KRB5_EXPORT_WITH_DES_CBC_40_SHA, TLS_KRB5_EXPORT_WITH_RC2_CBC_40_SHA, TLS_KRB5_EXPORT_WITH_RC4_40_SHA, TLS_KRB5_EXPORT_WITH_DES_CBC_40_MD5, TLS_KRB5_EXPORT_WITH_RC2_CBC_40_MD5, TLS_KRB5_EXPORT_WITH_RC4_40_MD5, TLS_PSK_WITH_NULL_SHA, TLS_DHE_PSK_WITH_NULL_SHA, TLS_RSA_PSK_WITH_NULL_SHA, TLS_RSA_WITH_AES_128_CBC_SHA, TLS_DH_DSS_WITH_AES_128_CBC_SHA, TLS_DH_RSA_WITH_AES_128_CBC_SHA, TLS_DHE_DSS_WITH_AES_128_CBC_SHA, TLS_DHE_RSA_WITH_AES_128_CBC_SHA, TLS_DH_anon_WITH_AES_128_CBC_SHA, TLS_RSA_WITH_AES_256_CBC_SHA, TLS_DH_DSS_WITH_AES_256_CBC_SHA, TLS_DH_RSA_WITH_AES_256_CBC_SHA, TLS_DHE_DSS_WITH_AES_256_CBC_SHA, TLS_DHE_RSA_WITH_AES_256_CBC_SHA, TLS_DH_anon_WITH_AES_256_CBC_SHA, TLS_RSA_WITH_NULL_SHA256, TLS_RSA_WITH_AES_128_CBC_SHA256, TLS_RSA_WITH_AES_256_CBC_SHA256, TLS_DH_DSS_WITH_AES_128_CBC_SHA256, TLS_DH_RSA_WITH_AES_128_CBC_SHA256, TLS_DHE_DSS_WITH_AES_128_CBC_SHA256, TLS_RSA_WITH_CAMELLIA_128_CBC_SHA, TLS_DH_DSS_WITH_CAMELLIA_128_CBC_SHA, TLS_DH_RSA_WITH_CAMELLIA_128_CBC_SHA, TLS_DHE_DSS_WITH_CAMELLIA_128_CBC_SHA, TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA, TLS_DH_anon_WITH_CAMELLIA_128_CBC_SHA, TLS_DHE_RSA_WITH_AES_128_CBC_SHA256, TLS_DH_DSS_WITH_AES_256_CBC_SHA256,

TLS_DH_RSA_WITH_AES_256_CBC_SHA256,
TLS_DHE_DSS_WITH_AES_256_CBC_SHA256,
TLS_DHE_RSA_WITH_AES_256_CBC_SHA256,
TLS_DH_anon_WITH_AES_128_CBC_SHA256,
TLS_DH_anon_WITH_AES_256_CBC_SHA256,
TLS_RSA_WITH_CAMELLIA_256_CBC_SHA,
TLS_DH_DSS_WITH_CAMELLIA_256_CBC_SHA,
TLS_DH_RSA_WITH_CAMELLIA_256_CBC_SHA,
TLS_DHE_DSS_WITH_CAMELLIA_256_CBC_SHA,
TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA,
TLS_DH_anon_WITH_CAMELLIA_256_CBC_SHA, TLS_PSK_WITH_RC4_128_SHA,
TLS_PSK_WITH_3DES_EDE_CBC_SHA, TLS_PSK_WITH_AES_128_CBC_SHA,
TLS_PSK_WITH_AES_256_CBC_SHA, TLS_DHE_PSK_WITH_RC4_128_SHA,
TLS_DHE_PSK_WITH_3DES_EDE_CBC_SHA, TLS_DHE_PSK_WITH_AES_128_CBC_SHA,
TLS_DHE_PSK_WITH_AES_256_CBC_SHA, TLS_RSA_PSK_WITH_RC4_128_SHA,
TLS_RSA_PSK_WITH_3DES_EDE_CBC_SHA, TLS_RSA_PSK_WITH_AES_128_CBC_SHA,
TLS_RSA_PSK_WITH_AES_256_CBC_SHA, TLS_RSA_WITH_SEED_CBC_SHA,
TLS_DH_DSS_WITH_SEED_CBC_SHA, TLS_DH_RSA_WITH_SEED_CBC_SHA,
TLS_DHE_DSS_WITH_SEED_CBC_SHA, TLS_DHE_RSA_WITH_SEED_CBC_SHA,
TLS_DH_anon_WITH_SEED_CBC_SHA, TLS_RSA_WITH_AES_128_GCM_SHA256,
TLS_RSA_WITH_AES_256_GCM_SHA384, TLS_DH_RSA_WITH_AES_128_GCM_SHA256,
TLS_DH_RSA_WITH_AES_256_GCM_SHA384,
TLS_DH_DSS_WITH_AES_128_GCM_SHA256,
TLS_DH_DSS_WITH_AES_256_GCM_SHA384,
TLS_DH_anon_WITH_AES_128_GCM_SHA256,
TLS_DH_anon_WITH_AES_256_GCM_SHA384, TLS_PSK_WITH_AES_128_GCM_SHA256,
TLS_PSK_WITH_AES_256_GCM_SHA384, TLS_RSA_PSK_WITH_AES_128_GCM_SHA256,
TLS_RSA_PSK_WITH_AES_256_GCM_SHA384, TLS_PSK_WITH_AES_128_CBC_SHA256,
TLS_PSK_WITH_AES_256_CBC_SHA384, TLS_PSK_WITH_NULL_SHA256,
TLS_PSK_WITH_NULL_SHA384, TLS_DHE_PSK_WITH_AES_128_CBC_SHA256,
TLS_DHE_PSK_WITH_AES_256_CBC_SHA384, TLS_DHE_PSK_WITH_NULL_SHA256,
TLS_DHE_PSK_WITH_NULL_SHA384, TLS_RSA_PSK_WITH_AES_128_CBC_SHA256,
TLS_RSA_PSK_WITH_AES_256_CBC_SHA384, TLS_RSA_PSK_WITH_NULL_SHA256,
TLS_RSA_PSK_WITH_NULL_SHA384, TLS_RSA_WITH_CAMELLIA_128_CBC_SHA256,
TLS_DH_DSS_WITH_CAMELLIA_128_CBC_SHA256,
TLS_DH_RSA_WITH_CAMELLIA_128_CBC_SHA256,
TLS_DHE_DSS_WITH_CAMELLIA_128_CBC_SHA256,
TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA256,
TLS_DH_anon_WITH_CAMELLIA_128_CBC_SHA256,
TLS_RSA_WITH_CAMELLIA_256_CBC_SHA256,
TLS_DH_DSS_WITH_CAMELLIA_256_CBC_SHA256,
TLS_DH_RSA_WITH_CAMELLIA_256_CBC_SHA256,
TLS_DHE_DSS_WITH_CAMELLIA_256_CBC_SHA256,
TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA256,
TLS_DH_anon_WITH_CAMELLIA_256_CBC_SHA256,
TLS_EMPTY_RENEGOTIATION_INFO_SCSV, TLS_ECDH_ECDSA_WITH_NULL_SHA,
TLS_ECDH_ECDSA_WITH_RC4_128_SHA,
TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA,

TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA,
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA, TLS_ECDHE_ECDSA_WITH_NULL_SHA,
TLS_ECDHE_ECDSA_WITH_RC4_128_SHA,
TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA,
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA,
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA, TLS_ECDH_RSA_WITH_NULL_SHA,
TLS_ECDH_RSA_WITH_RC4_128_SHA, TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA,
TLS_ECDH_RSA_WITH_AES_128_CBC_SHA, TLS_ECDH_RSA_WITH_AES_256_CBC_SHA,
TLS_ECDHE_RSA_WITH_NULL_SHA, TLS_ECDHE_RSA_WITH_RC4_128_SHA,
TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA,
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA,
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA, TLS_ECDH_anon_WITH_NULL_SHA,
TLS_ECDH_anon_WITH_RC4_128_SHA, TLS_ECDH_anon_WITH_3DES_EDE_CBC_SHA,
TLS_ECDH_anon_WITH_AES_128_CBC_SHA,
TLS_ECDH_anon_WITH_AES_256_CBC_SHA,
TLS_SRP_SHA_WITH_3DES_EDE_CBC_SHA,
TLS_SRP_SHA_RSA_WITH_3DES_EDE_CBC_SHA,
TLS_SRP_SHA_DSS_WITH_3DES_EDE_CBC_SHA,
TLS_SRP_SHA_WITH_AES_128_CBC_SHA,
TLS_SRP_SHA_RSA_WITH_AES_128_CBC_SHA,
TLS_SRP_SHA_DSS_WITH_AES_128_CBC_SHA,
TLS_SRP_SHA_WITH_AES_256_CBC_SHA,
TLS_SRP_SHA_RSA_WITH_AES_256_CBC_SHA,
TLS_SRP_SHA_DSS_WITH_AES_256_CBC_SHA,
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256,
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384,
TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256,
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA384,
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256,
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384,
TLS_ECDH_RSA_WITH_AES_128_CBC_SHA256,
TLS_ECDH_RSA_WITH_AES_256_CBC_SHA384,
TLS_ECDH_ECDSA_WITH_AES_128_GCM_SHA256,
TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384,
TLS_ECDH_RSA_WITH_AES_128_GCM_SHA256,
TLS_ECDH_RSA_WITH_AES_256_GCM_SHA384, TLS_ECDHE_PSK_WITH_RC4_128_SHA,
TLS_ECDHE_PSK_WITH_3DES_EDE_CBC_SHA,
TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA,
TLS_ECDHE_PSK_WITH_AES_256_CBC_SHA,
TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256,
TLS_ECDHE_PSK_WITH_AES_256_CBC_SHA384, TLS_ECDHE_PSK_WITH_NULL_SHA,
TLS_ECDHE_PSK_WITH_NULL_SHA256, TLS_ECDHE_PSK_WITH_NULL_SHA384,
TLS_RSA_WITH_ARIA_128_CBC_SHA256, TLS_RSA_WITH_ARIA_256_CBC_SHA384,
TLS_DH_DSS_WITH_ARIA_128_CBC_SHA256,
TLS_DH_DSS_WITH_ARIA_256_CBC_SHA384,
TLS_DH_RSA_WITH_ARIA_128_CBC_SHA256,
TLS_DH_RSA_WITH_ARIA_256_CBC_SHA384,
TLS_DHE_DSS_WITH_ARIA_128_CBC_SHA256,

TLS_DHE_DSS_WITH_ARIA_256_CBC_SHA384,
TLS_DHE_RSA_WITH_ARIA_128_CBC_SHA256,
TLS_DHE_RSA_WITH_ARIA_256_CBC_SHA384,
TLS_DH_anon_WITH_ARIA_128_CBC_SHA256,
TLS_DH_anon_WITH_ARIA_256_CBC_SHA384,
TLS_ECDHE_ECDSA_WITH_ARIA_128_CBC_SHA256,
TLS_ECDHE_ECDSA_WITH_ARIA_256_CBC_SHA384,
TLS_ECDH_ECDSA_WITH_ARIA_128_CBC_SHA256,
TLS_ECDH_ECDSA_WITH_ARIA_256_CBC_SHA384,
TLS_ECDHE_RSA_WITH_ARIA_128_CBC_SHA256,
TLS_ECDHE_RSA_WITH_ARIA_256_CBC_SHA384,
TLS_ECDH_RSA_WITH_ARIA_128_CBC_SHA256,
TLS_ECDH_RSA_WITH_ARIA_256_CBC_SHA384,
TLS_RSA_WITH_ARIA_128_GCM_SHA256, TLS_RSA_WITH_ARIA_256_GCM_SHA384,
TLS_DH_RSA_WITH_ARIA_128_GCM_SHA256,
TLS_DH_RSA_WITH_ARIA_256_GCM_SHA384,
TLS_DH_DSS_WITH_ARIA_128_GCM_SHA256,
TLS_DH_DSS_WITH_ARIA_256_GCM_SHA384,
TLS_DH_anon_WITH_ARIA_128_GCM_SHA256,
TLS_DH_anon_WITH_ARIA_256_GCM_SHA384,
TLS_ECDH_ECDSA_WITH_ARIA_128_GCM_SHA256,
TLS_ECDH_ECDSA_WITH_ARIA_256_GCM_SHA384,
TLS_ECDH_RSA_WITH_ARIA_128_GCM_SHA256,
TLS_ECDH_RSA_WITH_ARIA_256_GCM_SHA384,
TLS_PSK_WITH_ARIA_128_CBC_SHA256, TLS_PSK_WITH_ARIA_256_CBC_SHA384,
TLS_DHE_PSK_WITH_ARIA_128_CBC_SHA256,
TLS_DHE_PSK_WITH_ARIA_256_CBC_SHA384,
TLS_RSA_PSK_WITH_ARIA_128_CBC_SHA256,
TLS_RSA_PSK_WITH_ARIA_256_CBC_SHA384,
TLS_PSK_WITH_ARIA_128_GCM_SHA256, TLS_PSK_WITH_ARIA_256_GCM_SHA384,
TLS_RSA_PSK_WITH_ARIA_128_GCM_SHA256,
TLS_RSA_PSK_WITH_ARIA_256_GCM_SHA384,
TLS_ECDHE_PSK_WITH_ARIA_128_CBC_SHA256,
TLS_ECDHE_PSK_WITH_ARIA_256_CBC_SHA384,
TLS_ECDHE_ECDSA_WITH_CAMELLIA_128_CBC_SHA256,
TLS_ECDHE_ECDSA_WITH_CAMELLIA_256_CBC_SHA384,
TLS_ECDH_ECDSA_WITH_CAMELLIA_128_CBC_SHA256,
TLS_ECDH_ECDSA_WITH_CAMELLIA_256_CBC_SHA384,
TLS_ECDHE_RSA_WITH_CAMELLIA_128_CBC_SHA256,
TLS_ECDHE_RSA_WITH_CAMELLIA_256_CBC_SHA384,
TLS_ECDH_RSA_WITH_CAMELLIA_128_CBC_SHA256,
TLS_ECDH_RSA_WITH_CAMELLIA_256_CBC_SHA384,
TLS_RSA_WITH_CAMELLIA_128_GCM_SHA256,
TLS_RSA_WITH_CAMELLIA_256_GCM_SHA384,
TLS_DH_RSA_WITH_CAMELLIA_128_GCM_SHA256,
TLS_DH_RSA_WITH_CAMELLIA_256_GCM_SHA384,
TLS_DH_DSS_WITH_CAMELLIA_128_GCM_SHA256,
TLS_DH_DSS_WITH_CAMELLIA_256_GCM_SHA384,

TLS_DH_anon_WITH_CAMELLIA_128_GCM_SHA256,
TLS_DH_anon_WITH_CAMELLIA_256_GCM_SHA384,
TLS_ECDH_ECDSA_WITH_CAMELLIA_128_GCM_SHA256,
TLS_ECDH_ECDSA_WITH_CAMELLIA_256_GCM_SHA384,
TLS_ECDH_RSA_WITH_CAMELLIA_128_GCM_SHA256,
TLS_ECDH_RSA_WITH_CAMELLIA_256_GCM_SHA384,
TLS_PSK_WITH_CAMELLIA_128_GCM_SHA256,
TLS_PSK_WITH_CAMELLIA_256_GCM_SHA384,
TLS_RSA_PSK_WITH_CAMELLIA_128_GCM_SHA256,
TLS_RSA_PSK_WITH_CAMELLIA_256_GCM_SHA384,
TLS_PSK_WITH_CAMELLIA_128_CBC_SHA256,
TLS_PSK_WITH_CAMELLIA_256_CBC_SHA384,
TLS_DHE_PSK_WITH_CAMELLIA_128_CBC_SHA256,
TLS_DHE_PSK_WITH_CAMELLIA_256_CBC_SHA384,
TLS_RSA_PSK_WITH_CAMELLIA_128_CBC_SHA256,
TLS_RSA_PSK_WITH_CAMELLIA_256_CBC_SHA384,
TLS_ECDHE_PSK_WITH_CAMELLIA_128_CBC_SHA256,
TLS_ECDHE_PSK_WITH_CAMELLIA_256_CBC_SHA384, TLS_RSA_WITH_AES_128_CCM,
TLS_RSA_WITH_AES_256_CCM, TLS_RSA_WITH_AES_128_CCM_8,
TLS_RSA_WITH_AES_256_CCM_8, TLS_PSK_WITH_AES_128_CCM,
TLS_PSK_WITH_AES_256_CCM, TLS_PSK_WITH_AES_128_CCM_8,
TLS_PSK_WITH_AES_256_CCM_8.

Note: This list was assembled from the set of registered TLS cipher suites at the time of writing. This list includes those cipher suites that do not offer an ephemeral key exchange and those that are based on the TLS null, stream or block cipher type (as defined in Section 6.2.3 of [TLS12]). Additional cipher suites with these properties could be defined; these would not be explicitly prohibited.

Appendix B. Change Log

This section is to be removed by RFC Editor before publication.

B.1. Since draft-ietf-httpbis-http2-15

Enabled the sending of PRIORITY for any stream state.

Added a cipher suite blacklist and made several changes to the TLS usage section.

B.2. Since draft-ietf-httpbis-http2-14

Renamed Not Authoritative status code to Misdirected Request.

Added HTTP_1_1_REQUIRED error code.

B.3. Since draft-ietf-httpbis-http2-13

Pseudo-header fields are now required to appear strictly before regular ones.

Restored lxx series status codes, except 101.

Changed frame length field 24-bits. Expanded frame header to 9 octets. Added a setting to limit the damage.

Added a setting to advise peers of header set size limits.

Removed segments.

Made non-semantic-bearing HEADERS frames illegal in the HTTP mapping.

B.4. Since draft-ietf-httpbis-http2-12

Restored extensibility options.

Restricting TLS cipher suites to AEAD only.

Removing Content-Encoding requirements.

Permitting the use of PRIORITY after stream close.

Removed ALTSVC frame.

Removed BLOCKED frame.

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

Removed per-frame GZIP compression.

B.5. Since draft-ietf-httpbis-http2-11

Added BLOCKED frame (at risk).

Simplified priority scheme.

Added DATA per-frame GZIP compression.

B.6. Since draft-ietf-httpbis-http2-10

Changed "connection header" to "connection preface" to avoid confusion.

Added dependency-based stream prioritization.

Added "h2c" identifier to distinguish between cleartext and secured HTTP/2.

Adding missing padding to PUSH_PROMISE.

Integrate ALTSVC frame and supporting text.

Dropping requirement on "deflate" Content-Encoding.

Improving security considerations around use of compression.

B.7. Since draft-ietf-httpbis-http2-09

Adding padding for data frames.

Renumbering frame types, error codes, and settings.

Adding INADEQUATE_SECURITY error code.

Updating TLS usage requirements to 1.2; forbidding TLS compression.

Removing extensibility for frames and settings.

Changing setting identifier size.

Removing the ability to disable flow control.

Changing the protocol identification token to "h2".

Changing the use of :authority to make it optional and to allow userinfo in non-HTTP cases.

Allowing split on 0x0 for Cookie.

Reserved PRI method in HTTP/1.1 to avoid possible future collisions.

B.8. Since draft-ietf-httpbis-http2-08

Added cookie crumbling for more efficient header compression.

Added header field ordering with the value-concatenation mechanism.

B.9. Since draft-ietf-httpbis-http2-07

Marked draft for implementation.

B.10. Since draft-ietf-httpbis-http2-06

Adding definition for CONNECT method.

Constraining the use of push to safe, cacheable methods with no request body.

Changing from :host to :authority to remove any potential confusion.

Adding setting for header compression table size.

Adding settings acknowledgement.

Removing unnecessary and potentially problematic flags from CONTINUATION.

Added denial of service considerations.

B.11. Since draft-ietf-httpbis-http2-05

Marking the draft ready for implementation.

Renumbering END_PUSH_PROMISE flag.

Editorial clarifications and changes.

B.12. Since draft-ietf-httpbis-http2-04

Added CONTINUATION frame for HEADERS and PUSH_PROMISE.

PUSH_PROMISE is no longer implicitly prohibited if SETTINGS_MAX_CONCURRENT_STREAMS is zero.

Push expanded to allow all safe methods without a request body.

Clarified the use of HTTP header fields in requests and responses. Prohibited HTTP/1.1 hop-by-hop header fields.

Requiring that intermediaries not forward requests with missing or illegal routing :-headers.

Clarified requirements around handling different frames after stream close, stream reset and GOAWAY.

Added more specific prohibitions for sending of different frame types in various stream states.

Making the last received setting value the effective value.

Clarified requirements on TLS version, extension and ciphers.

B.13. Since draft-ietf-httpbis-http2-03

Committed major restructuring atrocities.

Added reference to first header compression draft.

Added more formal description of frame lifecycle.

Moved END_STREAM (renamed from FINAL) back to HEADERS/DATA.

Removed HEADERS+PRIORITY, added optional priority to HEADERS frame.

Added PRIORITY frame.

B.14. Since draft-ietf-httpbis-http2-02

Added continuations to frames carrying header blocks.

Replaced use of "session" with "connection" to avoid confusion with other HTTP stateful concepts, like cookies.

Removed "message".

Switched to TLS ALPN from NPN.

Editorial changes.

B.15. Since draft-ietf-httpbis-http2-01

Added IANA considerations section for frame types, error codes and settings.

Removed data frame compression.

Added PUSH_PROMISE.

Added globally applicable flags to framing.

Removed zlib-based header compression mechanism.

Updated references.

Clarified stream identifier reuse.

Removed CREDENTIALS frame and associated mechanisms.

Added advice against naive implementation of flow control.

Added session header section.

Restructured frame header. Removed distinction between data and control frames.

Altered flow control properties to include session-level limits.

Added note on cacheability of pushed resources and multiple tenant servers.

Changed protocol label form based on discussions.

B.16. Since draft-ietf-httpbis-http2-00

Changed title throughout.

Removed section on Incompatibilities with SPDY draft#2.

Changed INTERNAL_ERROR on GOAWAY to have a value of 2 [6].

Replaced abstract and introduction.

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

Removed unused references.

Added flow control principles (Section 5.2.1) based on [7].

B.17. 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)
Mozilla
331 E Evelyn Street
Mountain View, CA 94041
US

EMail: martin.thomson@gmail.com

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 4, 2014

S. Friedl
Cisco Systems, Inc.
A. Popov
Microsoft Corp.
A. Langley
Google Inc.
E. Stephan
Orange
March 3, 2014

Transport Layer Security (TLS) Application Layer Protocol Negotiation
Extension
draft-ietf-tls-applayerprotoneg-05

Abstract

This document describes a Transport Layer Security (TLS) extension for application layer protocol negotiation within the TLS handshake. For instances in which the TLS connection is established over a well known TCP or UDP port not associated with the desired application layer protocol, this extension allows the application layer to negotiate which protocol will be used within the TLS connection.

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 September 4, 2014.

Copyright Notice

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

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents

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

Table of Contents

1. Introduction	2
2. Requirements Language	3
3. Application Layer Protocol Negotiation	3
3.1. The Application Layer Protocol Negotiation Extension . .	3
3.2. Protocol Selection	5
4. Design Considerations	5
5. Security Considerations	6
6. IANA Considerations	6
7. Acknowledgements	7
8. References	8
8.1. Normative References	8
8.2. Informative References	8
Authors' Addresses	8

1. Introduction

Increasingly, application layer protocols are encapsulated in the TLS security protocol [RFC5246]. This encapsulation enables applications to use the existing, secure communications links already present on port 443 across virtually the entire global IP infrastructure.

When multiple application protocols are supported on a single server-side port number, such as port 443, the client and the server need to negotiate an application protocol for use with each connection. It is desirable to accomplish this negotiation without adding network round-trips between the client and the server, as each round-trip will degrade an end-user's experience. Further, it would be advantageous to allow certificate selection based on the negotiated application protocol.

This document specifies a TLS extension which permits the application layer to negotiate protocol selection within the TLS handshake. This work was requested by the HTTPbis WG to address the negotiation of HTTP version ([RFC2616], [I-D.ietf-httpbis-http2]) over TLS, however ALPN facilitates negotiation of arbitrary application layer protocols.

With ALPN, the client sends the list of supported application protocols as part of the TLS ClientHello message. The server chooses a protocol and sends the selected protocol as part of the TLS ServerHello message. The application protocol negotiation can thus be accomplished within the TLS handshake, without adding network round-trips, and allows the server to associate a different certificate with each application protocol, if desired.

2. Requirements Language

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 [RFC2119].

3. Application Layer Protocol Negotiation

3.1. The Application Layer Protocol Negotiation Extension

A new extension type ("application_layer_protocol_negotiation(16)") is defined and MAY be included by the client in its "ClientHello" message.

```
enum {  
    application_layer_protocol_negotiation(16), (65535)  
} ExtensionType;
```

The "extension_data" field of the ("application_layer_protocol_negotiation(16)") extension SHALL contain a "ProtocolNameList" value.

```
opaque ProtocolName<1..2^8-1>;
```

```
struct {  
    ProtocolName protocol_name_list<2..2^16-1>  
} ProtocolNameList;
```

"ProtocolNameList" contains the list of protocols advertised by the client, in descending order of preference. Protocols are named by IANA registered, opaque, non-empty byte strings, as described further in Section 6 "IANA Considerations" of this document. Empty strings MUST NOT be included and byte strings MUST NOT be truncated.

Servers that receive a client hello containing the "application_layer_protocol_negotiation" extension, MAY return a suitable protocol selection response to the client. The server will ignore any protocol name that it does not recognize. A new ServerHello extension type ("application_layer_protocol_negotiation(16)") MAY be returned to the

client within the extended ServerHello message. The "extension_data" field of the ("application_layer_protocol_negotiation(16)") extension is structured the same as described above for the client "extension_data", except that the "ProtocolNameList" MUST contain exactly one "ProtocolName".

Therefore, a full handshake with the "application_layer_protocol_negotiation" extension in the ClientHello and ServerHello messages has the following flow (contrast with section 7.3 of [RFC5246]):

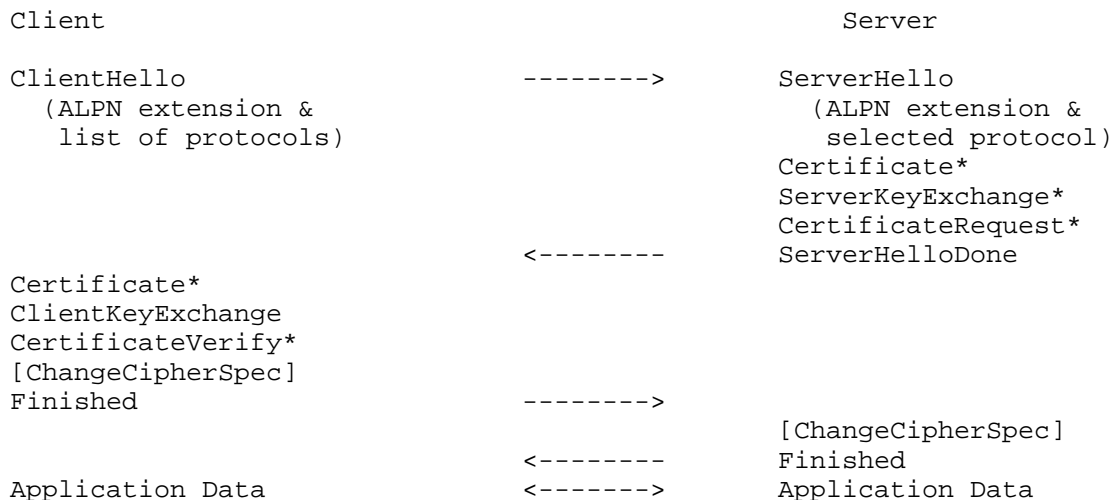


Figure 1

An abbreviated handshake with the "application_layer_protocol_negotiation" extension has the following flow:

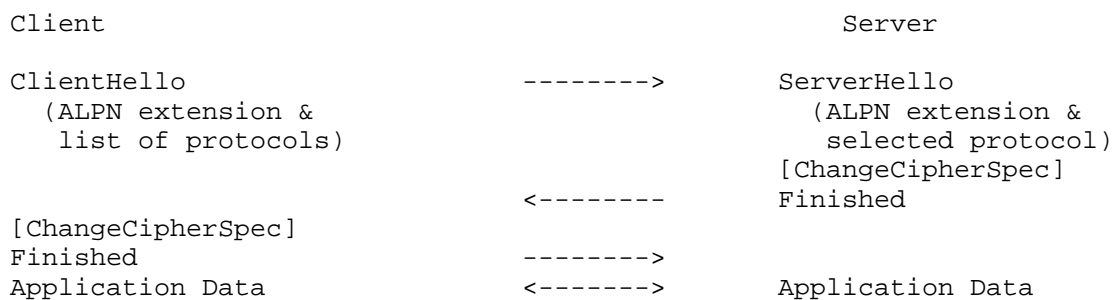


Figure 2

Unlike many other TLS extensions, this extension does not establish properties of the session, only of the connection. When session resumption or session tickets [RFC5077] are used, the previous contents of this extension are irrelevant and only the values in the new handshake messages are considered.

3.2. Protocol Selection

It is expected that a server will have a list of protocols that it supports, in preference order, and will only select a protocol if the client supports it. In that case, the server SHOULD select the most highly preferred protocol it supports which is also advertised by the client. In the event that the server supports no protocols that the client advertises, then the server SHALL respond with a fatal "no_application_protocol" alert.

```
enum {  
    no_application_protocol(120),  
    (255)  
} AlertDescription;
```

The protocol identified in the "application_layer_protocol_negotiation" extension type in the ServerHello SHALL be definitive for the connection, until renegotiated. The server SHALL NOT respond with a selected protocol and subsequently use a different protocol for application data exchange.

4. Design Considerations

The ALPN extension is intended to follow the typical design of TLS protocol extensions. Specifically, the negotiation is performed entirely within the client/server hello exchange in accordance with established TLS architecture. The "application_layer_protocol_negotiation" ServerHello extension is intended to be definitive for the connection (until the connection is renegotiated) and is sent in plaintext to permit network elements to provide differentiated service for the connection when the TCP or UDP port number is not definitive for the application layer protocol to be used in the connection. By placing ownership of protocol selection on the server, ALPN facilitates scenarios in which certificate selection or connection rerouting may be based on the negotiated protocol.

Finally, by managing protocol selection in the clear as part of the handshake, ALPN avoids introducing false confidence with respect to the ability to hide the negotiated protocol in advance of establishing the connection. If hiding the protocol is required,

then renegotiation after connection establishment, which would provide true TLS security guarantees, would be a preferred methodology.

5. Security Considerations

The ALPN extension does not impact the security of TLS session establishment or application data exchange. ALPN serves to provide an externally visible marker for the application layer protocol associated with the TLS connection. Historically, the application layer protocol associated with a connection could be ascertained from the TCP or UDP port number in use.

Implementers and document editors who intend to extend the protocol identifier registry by adding new protocol identifiers should consider that in TLS versions 1.2 and below the client sends these identifiers in the clear, and should also consider that for at least the next decade, it is expected that browsers would normally use these earlier versions of TLS in the initial ClientHello.

Care must be taken when such identifiers may leak personally identifiable information, or when such leakage may lead to profiling, or to leaking of sensitive information. If any of these apply to this new protocol identifier, the identifier SHOULD NOT be used in TLS configurations where it would be visible in the clear, and documents specifying such protocol identifiers SHOULD recommend against such unsafe use.

6. IANA Considerations

The IANA has updated its Registry of TLS ExtensionType Values to include the following entry:

16 application_layer_protocol_negotiation

This document establishes a registry for protocol identifiers entitled "Application Layer Protocol Negotiation (ALPN) Protocol IDs" under the existing "Transport Layer Security (TLS)" heading.

Entries in this registry require the following fields:

- o Protocol: The name of the protocol.
- o Identification Sequence: The precise set of octet values that identifies the protocol. This could be the UTF-8 encoding [RFC3629] of the protocol name.

- o Specification: A reference to a specification that defines the protocol.

This registry operates under the "Expert Review" policy as defined in [RFC5226]. The designated expert is advised to encourage the inclusion of a reference to a permanent and readily available specification that enables the creation of interoperable implementations of the identified protocol.

An initial set of registrations for this registry follows:

Protocol: HTTP/1.1

Identification Sequence: 0x68 0x74 0x74 0x70 0x2f 0x31 0x2e 0x31
("http/1.1")

Specification: <http://tools.ietf.org/html/rfc2616>

Protocol: SPDY/1

Identification Sequence: 0x73 0x70 0x64 0x79 0x2f 0x31 ("spdy/1")

Specification: <http://dev.chromium.org/spdy/spdy-protocol/spdy-protocol-draft1>

Protocol: SPDY/2

Identification Sequence: 0x73 0x70 0x64 0x79 0x2f 0x32 ("spdy/2")

Specification: <http://dev.chromium.org/spdy/spdy-protocol/spdy-protocol-draft2>

Protocol: SPDY/3

Identification Sequence: 0x73 0x70 0x64 0x79 0x2f 0x33 ("spdy/3")

Specification: <http://dev.chromium.org/spdy/spdy-protocol/spdy-protocol-draft3>

7. Acknowledgements

This document benefitted specifically from the NPN extension draft authored by Adam Langley and from discussions with Tom Wesselman and Cullen Jennings both of Cisco.

8. References

8.1. Normative References

- [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.
- [RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.
- [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

8.2. Informative References

- [I-D.ietf-httpbis-http2] Belshé, M., Peon, R., and M. Thomson, "Hypertext Transfer Protocol version 2", draft-ietf-httpbis-http2-10 (work in progress), February 2014.
- [RFC5077] Salowey, J., Zhou, H., Eronen, P., and H. Tschofenig, "Transport Layer Security (TLS) Session Resumption without Server-Side State", RFC 5077, January 2008.

Authors' Addresses

Stephan Friedl
Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134
USA

Phone: (720)562-6785
Email: sfriedl@cisco.com

Andrei Popov
Microsoft Corp.
One Microsoft Way
Redmond, WA 98052
USA

Email: andreipo@microsoft.com

Adam Langley
Google Inc.
USA

Email: agl@google.com

Emile Stephan
Orange
2 avenue Pierre Marzin
Lannion F-22307
France

Email: emile.stephan@orange.com

Network Working Group
Internet Draft
Intended status: Standards Track
Expires: August 2014

O. Mazahir
D. Thaler
M. Cox
G. Montenegro
Microsoft Corporation
14 February 2014

Deterministic URI Encoding
draft-montenegro-httpbis-uri-encoding-00

Abstract

The "http" and "https" URI schemes do not have a fixed character encoding. This document defines HTTP headers to enable an explicit indication of the character encoding.

Status of this Memo

This Internet-Draft is submitted to IETF in full conformance with the provisions of BCP 78 and BCP 79. This document may contain material from IETF Documents or IETF Contributions published or made publicly available before November 10, 2008. The person(s) controlling the copyright in some of this material may not have granted the IETF Trust the right to allow modifications of such material outside the IETF Standards Process. Without obtaining an adequate license from the person(s) controlling the copyright in such materials, this document may not be modified outside the IETF Standards Process, and derivative works of it may not be created outside the IETF Standards Process, except to format it for publication as an RFC or to translate it into languages other than English.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

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."

The list of current Internet-Drafts can be accessed at <http://www.ietf.org/ietf/lid-abstracts.txt>.

The list of Internet-Draft Shadow Directories can be accessed at <http://www.ietf.org/shadow.html>.

This Internet-Draft will expire on August, 2014.

Copyright

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

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

Table of Contents

1. Introduction.....	2
1.1. Requirements Language.....	3
2. URI Path and Query Encoding Headers.....	3
3. IANA Considerations.....	4
3.1. URI-Path-Encoding.....	4
3.2. URI-Query-Encoding.....	4
4. Security Considerations.....	5
5. Acknowledgments.....	5
6. References.....	5
6.1. Normative References.....	5
6.2. Informative References.....	5
7. Author's Addresses.....	6

1. Introduction

The "http" and "https" URI schemes don't have a fixed character encoding. The URI RFC [RFC3986] talks about the generic syntax for URI components:

- . Legacy URI components (before 2005) tend to use UTF-8 "or some other superset of the US-ASCII character encoding"
- . New schemes (after 2005) use UTF-8 with percent encoding for reserved characters.

The first bullet explains why the character encoding for "http" and "https" URIs is not deterministic. This is particularly
Mazahir, et. al. [Page 2]

problematic when parsing URIs at the server side or at intermediate proxies (e.g., when looking for a cache hit).

URI's have different components with different character encoding issues.

Per the IDNA rules in [RFC5890], the host component is encoded using A-labels.

There is more non-determinism with respect to the path and query components. Furthermore, these two components are not necessarily encoded the same way [Handbook].

This document defines HTTP headers that explicitly state the character encoding for the path and query components.

1.1. Requirements Language

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].

2. URI Path and Query Encoding Headers

The URI Path encoding is conveyed in the following header:

```
URI-Path-Encoding    = "URI-Path-Encoding" ":" 1charset
```

The URI Query encoding is conveyed in the following header:

```
URI-Query-Encoding   = "URI-Query-Encoding" ":" 1charset
```

charset is defined in section 3.4 of [RFC2616]. The expected value indicates the character encoding for the path or query component in the URI prior to percent encoding. (A value of UTF-8 does not mean that the URI carries raw UTF-8.)

If the user agent is certain that the path component was formed from percent-encoded UTF-8, it sets the header as follows:

```
URI-Path-Encoding: UTF-8
```

Similarly, for the query component:

```
URI-Query-Encoding: UTF-8
```

This signals that the query component in the URI is in UTF-8 with percent encoding.

Absence of the URI-Path-Encoding or URI-Query-Encoding header is equivalent to the legacy situation of non-determinism with respect to the path or query component, respectively, as mentioned above in section 1.

Likewise, if the URI-Path-Encoding or URI-Query-Encoding header is set to an invalid value or unrecognized charset, this is equivalent to the legacy situation of non-determinism with respect to the path or query component, respectively, mentioned above in section 1.

3. IANA Considerations

IANA is requested to add these headers to the "Permanent Message Header Field Names" registry. Per [RFC3864], the template for these headers is specified below.

3.1. URI-Path-Encoding

Applicable protocol: http

Status: standard

Author/change controller:

IETF (iesg@ietf.org)

Specification document(s):

This document.

3.2. URI-Query-Encoding

Applicable protocol: http

Status: standard

Author/change controller:

IETF (iesg@ietf.org)

Specification document(s):

This document.

4. Security Considerations

Due to the non-deterministic character encoding of URI's, URI parsing at servers or proxies currently may involve trying different possible character encodings searching for a match. This represents a potential attack vector [RFC6943]. The headers proposed in this document could be used to reduce the attack surface by enabling a more explicit interpretation of the data within a URI, thus preventing unintended consequences.

5. Acknowledgments

Thanks to Ivan Pashov and Wade Hilmo for useful discussions in this space.

This document was prepared using 2-Word-v2.0.template.doc.

6. References

6.1. Normative References

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

6.2. Informative References

- [Handbook] Zalewski, M., "Browser Security Handbook, part 1", <http://code.google.com/p/browsersec/wiki/Part1>
- Mazahir, et. al. [Page 5]

March 2011.

- [RFC3864] Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, September 2004.
- [RFC5890] Klensin, J., "Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework", RFC 5890, August 2010.
- [RFC6943] Thaler, D., Ed., "Issues in Identifier Comparison for Security Purposes", RFC 6943, May 2013.

7. Author's Addresses

Osama Mazahir
Microsoft Corporation

Email: OsamaM@microsoft.com

Dave Thaler
Microsoft Corporation

Email: DThaler@microsoft.com

Matthew Cox
Microsoft Corporation

Email: MaCox@microsoft.com

Gabriel Montenegro
Microsoft Corporation

Phone:
Email: gabriel.montenegro@microsoft.com

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 22, 2014

M. Nottingham
Akamai
P. McManus
Mozilla
March 21, 2014

HTTP Alternative Services
draft-nottingham-httpbis-alt-svc-05

Abstract

This document specifies "alternative services" for HTTP, which allow an origin's resources to be authoritatively available at a separate network location, possibly accessed with a different protocol configuration.

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 September 22, 2014.

Copyright Notice

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

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

Table of Contents

1. Introduction	3
1.1. Notational Conventions	3
2. Alternative Services Concepts	4
2.1. Host Authentication	5
2.2. Alternative Service Caching	5
2.3. Requiring Server Name Indication	6
2.4. Using Alternative Services	6
3. The Alt-Svc HTTP Header Field	7
3.1. Caching Alt-Svc Header Field Values	8
4. The Service HTTP Header Field	8
5. The 4NN Not Authoritative HTTP Status Code	9
6. IANA Considerations	9
6.1. The Alt-Svc Message Header Field	9
6.2. The Service Message Header Field	10
6.3. The 4NN Not Authoritative HTTP Status Code	10
7. Security Considerations	10
7.1. Changing Ports	10
7.2. Changing Hosts	11
7.3. Changing Protocols	11
8. References	12
8.1. Normative References	12
8.2. Informative References	12
Appendix A. Acknowledgements	13
Authors' Addresses	13

1. Introduction

HTTP [I-D.ietf-httpbis-pl-messaging] conflates the identification of resources with their location. In other words, `http://` (and `https://`) URLs are used to both name and find things to interact with.

In some cases, it is desirable to separate these aspects; to be able to keep the same identifier for a resource, but interact with it using a different location on the network.

For example:

- o An origin server might wish to redirect a client to an alternative when it needs to go down for maintenance, or it has found an alternative in a location that is more local to the client.
- o An origin server might wish to offer access to its resources using a new protocol (such as HTTP/2 [I-D.ietf-httpbis-http2]) or one using improved security (such as TLS [RFC5246]).
- o An origin server might wish to segment its clients into groups of capabilities, such as those supporting SNI (see [RFC6066]) and those not supporting it, for operational purposes.

This specification defines a new concept in HTTP, "Alternative Services", that allows a resource to nominate additional means of interacting with it on the network. It defines a general framework for this in Section 2, along with a specific mechanism for discovering them using HTTP headers in Section 3.

It also introduces a new status code in Section 5, so that origin servers (or their nominated alternatives) can indicate that they are not authoritative for a given origin, in cases where the wrong location is used.

1.1. Notational Conventions

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 [RFC2119].

This document uses the Augmented BNF defined in [RFC5234] along with the "OWS", "DIGIT", "parameter", "uri-host", "port" and "delta-second" rules from [I-D.ietf-httpbis-pl-messaging], and uses the "#rule" extension defined in Section 7 of that document.

2. Alternative Services Concepts

This specification defines a new concept in HTTP, the "alternative service." When an origin (see [RFC6454]) has resources accessible through a different protocol / host / port combination, it is said to have an alternative service.

An alternative service can be used to interact with the resources on an origin server at a separate location on the network, possibly using a different protocol configuration. Alternative services are considered authoritative for an origin's resources, in the sense of [I-D.ietf-httpbis-pl-messaging] Section 9.1.

For example, an origin:

```
("http", "www.example.com", "80")
```

might declare that its resources are also accessible at the alternative service:

```
("h2", "new.example.com", "81")
```

By their nature, alternative services are explicitly at the granularity of an origin; i.e., they cannot be selectively applied to resources within an origin.

Alternative services do not replace or change the origin for any given resource; in general, they are not visible to the software "above" the access mechanism. The alternative service is essentially alternative routing information that can also be used to reach the origin in the same way that DNS CNAME or SRV records define routing information at the name resolution level. Each origin maps to a set of these routes - the default route is derived from origin itself and the other routes are introduced based on alternative-protocol information.

Furthermore, it is important to note that the first member of an alternative service tuple is different from the "scheme" component of an origin; it is more specific, identifying not only the major version of the protocol being used, but potentially communication options for that protocol.

This means that clients using an alternative service will change the host, port and protocol that they are using to fetch resources, but these changes MUST NOT be propagated to the application that is using HTTP; from that standpoint, the URI being accessed and all information derived from it (scheme, host, port) are the same as before.

Importantly, this includes its security context; in particular, when TLS [RFC5246] is in use, the alternative server will need to present a certificate for the origin's host name, not that of the alternative. Likewise, the Host header is still derived from the origin, not the alternative service (just as it would if a CNAME were being used).

The changes MAY, however, be made visible in debugging tools, consoles, etc.

Formally, an alternative service is identified by the combination of:

- o An ALPN protocol, as per [I-D.ietf-tls-applayerprotoneg]
- o A host, as per [RFC3986]
- o A port, as per [RFC3986]

Additionally, each alternative service MUST have:

- o A freshness lifetime, expressed in seconds; see Section 2.2

There are many ways that a client could discover the alternative service(s) associated with an origin.

2.1. Host Authentication

Clients MUST NOT use alternative services with a host other than the origin's without strong server authentication; this mitigates the attack described in Section 7.2. One way to achieve this is for the alternative to use TLS with a certificate that is valid for that origin.

For example, if the origin's host is "www.example.com" and an alternative is offered on "other.example.com" with the "h2" protocol, and the certificate offered is valid for "www.example.com", the client can use the alternative. However, if "other.example.com" is offered with the "h2c" protocol, the client cannot use it, because there is no mechanism in that protocol to establish strong server authentication.

Furthermore, this means that the HTTP Host header and the SNI information provided in TLS by the client will be that of the origin, not the alternative.

2.2. Alternative Service Caching

Mechanisms for discovering alternative services can associate a freshness lifetime with them; for example, the Alt-Svc header field uses the "ma" parameter.

Clients MAY choose to use an alternative service instead of the origin at any time when it is considered fresh; see Section 2.4 for specific recommendations.

Clients with existing connections to alternative services are not required to fall back to the origin when its freshness lifetime ends; i.e., the caching mechanism is intended for limiting how long an alternative service can be used for establishing new requests, not limiting the use of existing ones.

To mitigate risks associated with caching compromised values (see Section 7.2 for details), user agents SHOULD examine cached alternative services when they detect a change in network configuration, and remove any that could be compromised (for example, those whose association with the trust root is questionable). UAs that do not have a means of detecting network changes SHOULD place an upper bound on their lifetime.

2.3. Requiring Server Name Indication

A client must only use a TLS-based alternative service if the client also supports TLS Server Name Indication (SNI) [RFC6066]. This supports the conservation of IP addresses on the alternative service host.

2.4. Using Alternative Services

By their nature, alternative services are optional; clients are not required to use them. However, it is advantageous for clients to behave in a predictable way when they are used by servers (e.g., for load balancing).

Therefore, if a client becomes aware of an alternative service, the client SHOULD use that alternative service for all requests to the associated origin as soon as it is available, provided that the security properties of the alternative service protocol are desirable, as compared to the existing connection.

When a client uses an alternate service, it MUST emit the Service header field Section 4 on every request using that alternate service.

The client is not required to block requests; the origin's connection can be used until the alternative connection is established. However, if the security properties of the existing connection are weak (e.g. cleartext HTTP/1.1) then it might make sense to block until the new connection is fully available in order to avoid information leakage.

Furthermore, if the connection to the alternative service fails or is unresponsive, the client MAY fall back to using the origin. Note, however, that this could be the basis of a downgrade attack, thus losing any enhanced security properties of the alternative service.

3. The Alt-Svc HTTP Header Field

A HTTP(S) origin server can advertise the availability of alternative services (see Section 2) to clients by adding an Alt-Svc header field to responses.

```
Alt-Svc      = 1#( alternative *( OWS ";" OWS parameter ) )
alternative   = <"> protocol-id <"> "=" port
protocol-id   = <ALPN protocol identifier>
```

For example:

```
Alt-Svc: "http2 "=8000
```

This indicates that the "http2" protocol on the same host using the indicated port (in this case, 8000).

Alt-Svc MAY occur in any HTTP response message, regardless of the status code.

Alt-Svc does not allow advertisement of alternative services on other hosts, to protect against various header-based attacks.

It can, however, have multiple values:

```
Alt-Svc: "h2c "=8000, "h2 "=443
```

The value(s) advertised by Alt-Svc can be used by clients to open a new connection to one or more alternative services immediately, or simultaneously with subsequent requests on the same connection.

Intermediaries MUST NOT change or append Alt-Svc values.

Finally, note that while it may be technically possible to put content other than printable ASCII in a HTTP header, some implementations only support ASCII (or a superset of it) in header field values. Therefore, this field SHOULD NOT be used to convey protocol identifiers that are not printable ASCII, or those that contain quote characters.

3.1. Caching Alt-Svc Header Field Values

When an alternative service is advertised using Alt-Svc, it is considered fresh for 24 hours from generation of the message. This can be modified with the 'ma' (max-age) parameter;

```
Alt-Svc: "h2"=443;ma=3600
```

which indicates the number of seconds since the response was generated the alternative service is considered fresh for.

ma = delta-seconds

See [I-D.ietf-httpbis-p6-cache] Section 4.2.3 for details of determining response age. For example, a response:

```
HTTP/1.1 200 OK
Content-Type: text/html
Cache-Control: 600
Age: 30
Alt-Svc: "h2c"=8000; ma=60
```

indicates that an alternative service is available and usable for the next 60 seconds. However, the response has already been cached for 30 seconds (as per the Age header field value), so therefore the alternative service is only fresh for the 30 seconds from when this response was received, minus estimated transit time.

When an Alt-Svc response header is received from an origin, its value invalidates and replaces all cached alternative services for that origin.

See Section 2.2 for general requirements on caching alternative services.

Note that the freshness lifetime for HTTP caching (here, 600 seconds) does not affect caching of Alt-Svc values.

4. The Service HTTP Header Field

The Service HTTP header field is used in requests to indicate the identity of the alternate service in use, just as the Host header identifies the host and port of the origin.

Service = uri-host [":" port]

Service is intended to allow alternate services to detect loops,

differentiate traffic for purposes of load balancing, and generally to ensure that it is possible to identify the intended destination of traffic, since introducing this information after a protocol is in use has proven to be problematic.

When using an Alternate Service, clients **MUST** include a Service header in all requests. For example:

```
GET /thing
Host: origin.example.com
Service: alternate.example.net
User-Agent: Example/1.0
```

5. The 4NN Not Authoritative HTTP Status Code

The 4NN (Not Authoritative) status code indicates that the current origin server (usually, but not always an alternative service; see Section 2) is not authoritative for the requested resource, in the sense of [I-D.ietf-httpbis-pl-messaging], Section 9.1.

Clients receiving 4NN (Not Authoritative) from an alternative service **MUST** remove the corresponding entry from its alternative service cache (see Section 2.2) for that origin. Regardless of the idempotency of the request method, they **MAY** retry the request, either at another alternative server, or at the origin.

4NN (Not Authoritative) **MAY** carry an Alt-Svc header field.

This status code **MUST NOT** be generated by proxies.

A 4NN response is cacheable by default; i.e., unless otherwise indicated by the method definition or explicit cache controls (see Section 4.2.2 of [I-D.ietf-httpbis-p6-cache]).

6. IANA Considerations

6.1. The Alt-Svc Message Header Field

This document registers Alt-Svc in the Permanent Message Header Registry [RFC3864].

- o Header Field Name: Alt-Svc
- o Application Protocol: http
- o Status: standard

- o Author/Change Controller: IETF
- o Specification Document: [this document]
- o Related Information:

6.2. The Service Message Header Field

This document registers Alt-Svc in the Permanent Message Header Registry [RFC3864].

- o Header Field Name: Service
- o Application Protocol: http
- o Status: standard
- o Author/Change Controller: IETF
- o Specification Document: [this document]
- o Related Information:

6.3. The 4NN Not Authoritative HTTP Status Code

This document registers the 4NN (Not Authoritative) HTTP Status code [I-D.ietf-httpbis-p2-semantics].

- o Status Code: 4NN
- o Short Description: Not Authoritative
- o Specification: [this document], Section 5

7. Security Considerations

Identified security considerations should be enumerated in the appropriate documents depending on which proposals are accepted. Those listed below are generic to all uses of alternative services; more specific ones might be necessary.

7.1. Changing Ports

Using an alternative service implies accessing an origin's resources on an alternative port, at a minimum. An attacker that can inject alternative services and listen at the advertised port is therefore able to hijack an origin.

For example, an attacker that can add HTTP response header fields can redirect traffic to a different port on the same host using the Alt-Svc header field; if that port is under the attacker's control, they can thus masquerade as the HTTP server.

This risk can be mitigated by restricting the ability to advertise alternative services, and restricting who can open a port for listening on that host.

7.2. Changing Hosts

When the host is changed due to the use of an alternative service, it presents an opportunity for attackers to hijack communication to an origin.

For example, if an attacker can convince a user agent to send all traffic for "innocent.example.org" to "evil.example.com" by successfully associating it as an alternative service, they can masquerade as that origin. This can be done locally (see mitigations above) or remotely (e.g., by an intermediary as a man-in-the-middle attack).

This is the reason for the requirement in Section 2.1 that any alternative service with a host different to the origin's be strongly authenticated with the origin's identity; i.e., presenting a certificate for the origin proves that the alternative service is authorized to serve traffic for the origin.

However, this authorization is only as strong as the method used to authenticate the alternative service. In particular, there are well-known exploits to make an attacker's certificate appear as legitimate.

Alternative services could be used to persist such an attack; for example, an intermediary could man-in-the-middle TLS-protected communication to a target, and then direct all traffic to an alternative service with a large freshness lifetime, so that the user agent still directs traffic to the attacker even when not using the intermediary.

As a result, there is a requirement in Section 2.2 to examine cached alternative services when a network change is detected.

7.3. Changing Protocols

When the ALPN protocol is changed due to the use of an alternative service, the security properties of the new connection to the origin can be different from that of the "normal" connection to the origin, because the protocol identifier itself implies this.

For example, if a "https://" URI had a protocol advertised that does not use some form of end-to-end encryption (most likely, TLS), it violates the expectations for security that the URI scheme implies.

Therefore, clients cannot blindly use alternative services, but instead evaluate the option(s) presented to assure that security requirements and expectations (of specifications, implementations and

end users) are met.

8. References

8.1. Normative References

- [I-D.ietf-httpbis-pl-messaging]
Fielding, R. and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", draft-ietf-httpbis-pl-messaging-26 (work in progress), February 2014.
- [I-D.ietf-httpbis-p6-cache]
Fielding, R., Nottingham, M., and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Caching", draft-ietf-httpbis-p6-cache-26 (work in progress), February 2014.
- [I-D.ietf-tls-applayerprotoneg]
Friedl, S., Popov, A., Langley, A., and S. Emile, "Transport Layer Security (TLS) Application Layer Protocol Negotiation Extension", draft-ietf-tls-applayerprotoneg-05 (work in progress), March 2014.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.
- [RFC5234] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008.
- [RFC6066] Eastlake, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, January 2011.
- [RFC6454] Barth, A., "The Web Origin Concept", RFC 6454, December 2011.

8.2. Informative References

- [I-D.ietf-httpbis-http2]
Belshe, M., Peon, R., and M. Thomson, "Hypertext Transfer Protocol version 2", draft-ietf-httpbis-http2-10 (work in progress), February 2014.

- [I-D.ietf-httpbis-p2-semantic]
Fielding, R. and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content",
draft-ietf-httpbis-p2-semantic-26 (work in progress),
February 2014.
- [RFC3864] Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, September 2004.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

Appendix A. Acknowledgements

Thanks to Eliot Lear, Stephen Farrell, Guy Podjarny, Stephen Ludin, Erik Nygren, Paul Hoffman, Adam Langley, Will Chan and Richard Barnes for their feedback and suggestions.

The Alt-Svc header field was influenced by the design of the Alternative-Protocol header in SPDY.

Authors' Addresses

Mark Nottingham
Akamai

Email: mnot@mnot.net
URI: <http://www.mnot.net/>

Patrick McManus
Mozilla

Email: mcmanus@ducksong.com
URI: <https://mozillians.org/u/pmcmanus/>

