

HTTPbis Working Group
Internet-Draft
Intended status: Informational
Expires: August 10, 2014

J. Reschke
greenbytes
February 6, 2014

Initial Hypertext Transfer Protocol (HTTP)
Authentication Scheme Registrations
draft-ietf-httpbis-authscheme-registrations-10

Abstract

This document registers Hypertext Transfer Protocol (HTTP) authentication schemes which have been defined in RFCs before the IANA HTTP Authentication Scheme Registry was established.

Editorial Note (To be removed by RFC Editor)

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

The current issues list is at <http://trac.tools.ietf.org/wg/httpbis/trac/query?component=authscheme-registrations> and related documents (including fancy diffs) can be found at <http://tools.ietf.org/wg/httpbis/>.

The changes in this draft are summarized in Appendix A.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 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. Security Considerations | 3 |
| 3. IANA Considerations | 3 |
| 4. Normative References | 3 |
| Appendix A. Change Log (to be removed by RFC Editor before publication) | 4 |
| A.1. Since draft-ietf-httpbis-authscheme-registrations-08 | 4 |
| A.2. Since draft-ietf-httpbis-authscheme-registrations-09 | 4 |

1. Introduction

This document registers Hypertext Transfer Protocol (HTTP) authentication schemes which have been defined in RFCs before the IANA HTTP Authentication Scheme Registry was established.

2. Security Considerations

There are no security considerations related to the registration itself.

Security considerations applicable to the individual authentication schemes ought to be discussed in the specifications that define them.

3. IANA Considerations

The table below provides registrations of HTTP authentication schemes to be added to the IANA HTTP Authentication Scheme registry at <<http://www.iana.org/assignments/http-authschemes>> (see Section 5.1 of [draft-ietf-httpbis-p7-auth]).

| Authentication Scheme Name | Reference | Notes |
|----------------------------|-----------------------------|--|
| Basic | [RFC2617], Section 2 | This authentication scheme violates both HTTP semantics (being connection-oriented) and syntax (use of syntax incompatible with the WWW-Authenticate and Authorization header field syntax). |
| Bearer | [RFC6750] | |
| Digest | [RFC2617], Section 3 | |
| Negotiate | [RFC4559], Section 3 | |
| OAuth | [RFC5849], Section 3.5.1 | |

4. Normative References

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

- [RFC4559] Jaganathan, K., Zhu, L., and J. Brezak, "SPNEGO-based Kerberos and NTLM HTTP Authentication in Microsoft Windows", RFC 4559, June 2006.
- [RFC5849] Hammer-Lahav, E., "The OAuth 1.0 Protocol", RFC 5849, April 2010.
- [RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, October 2012.
- [draft-ietf-httpbis-p7-auth] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Authentication", draft-ietf-httpbis-p7-auth-26 (work in progress), February 2014.

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

Changes up to the IETF Last Call draft are summarized in <<http://trac.tools.ietf.org/html/draft-ietf-httpbis-authscheme-registrations-08#appendix-B>>.

A.1. Since draft-ietf-httpbis-authscheme-registrations-08

Closed issues:

- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/514>>: "registration tables should be inside IANA considerations"

Clarified the IANA action to say "add".

Updated httpbis reference.

A.2. Since draft-ietf-httpbis-authscheme-registrations-09

Closed issues:

- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/530>>: "draft-ietf-httpbis-authscheme-registrations-09"

Updated httpbis reference.

Author's Address

Julian F. Reschke
greenbytes GmbH
Hafenweg 16
Muenster, NW 48155
Germany

EMail: julian.reschke@greenbytes.de
URI: <http://greenbytes.de/tech/webdav/>

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.

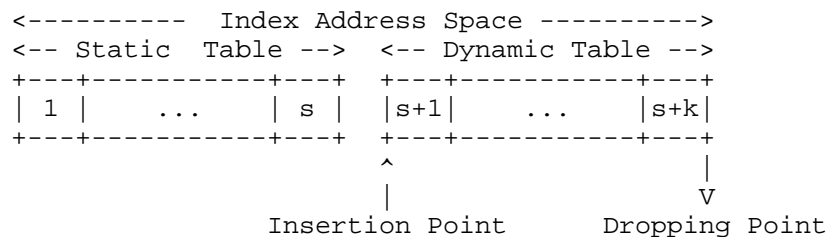


Figure 1: Index Address Space

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.

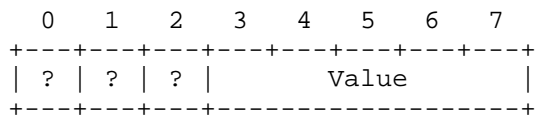


Figure 2: Integer Value Encoded within the Prefix (shown for N = 5)

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.

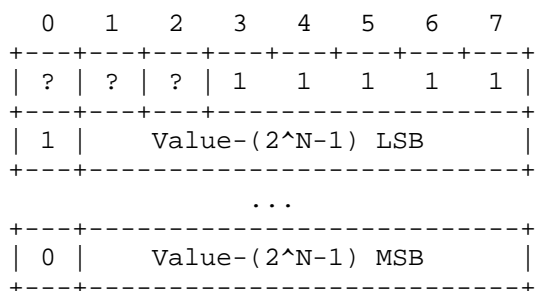


Figure 3: Integer Value Encoded after the Prefix (shown for N = 5)

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]).

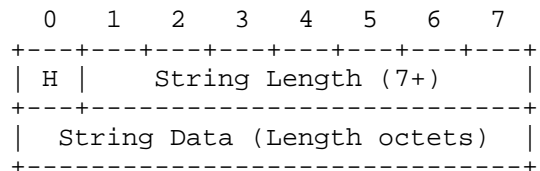


Figure 4: String Literal Representation

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.

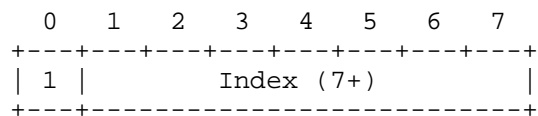


Figure 5: Indexed Header Field

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.

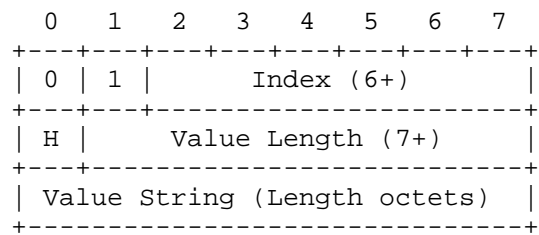


Figure 6: Literal Header Field with Incremental Indexing - Indexed Name

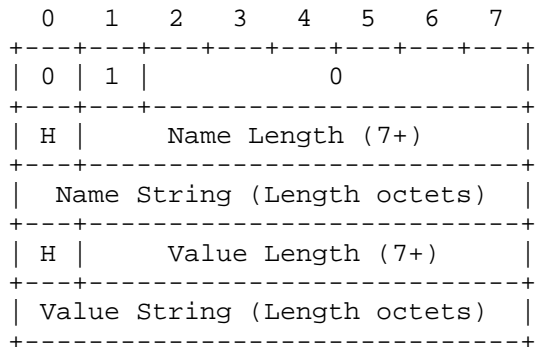


Figure 7: Literal Header Field with Incremental Indexing - New Name

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.

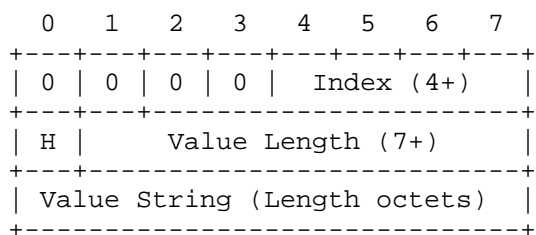


Figure 8: Literal Header Field without Indexing - Indexed Name

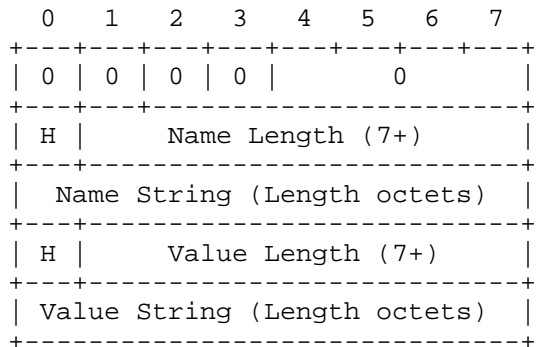


Figure 9: Literal Header Field without Indexing - New Name

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.

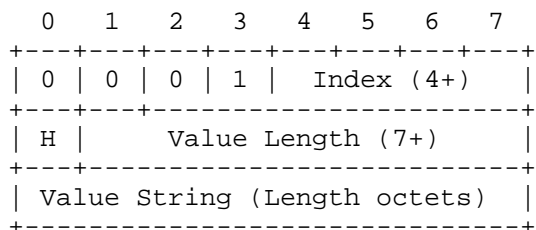


Figure 10: Literal Header Field never Indexed - Indexed Name

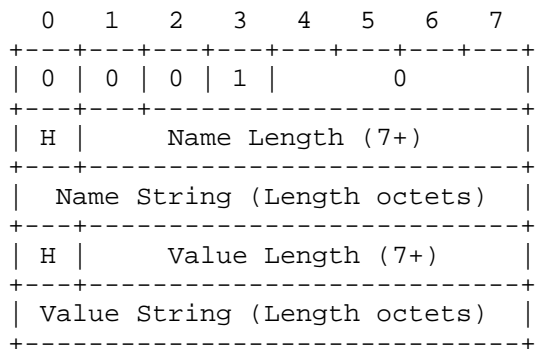


Figure 11: Literal Header Field never Indexed - New Name

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.

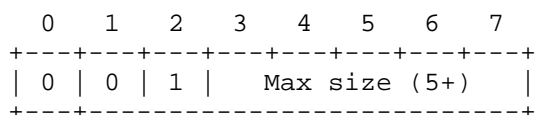


Figure 12: Maximum Dynamic Table Size Change

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 | 11111110 | 1100 | | fffec | [20] |
| (225) | 11111111 | 11111111 | 11110011 | | fffff3 | [24] |
| (226) | 11111111 | 11111110 | 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 | | 3fffea | [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 |
| 86 | -> :method: GET == Indexed - Add == idx = 6 |
| 84 | -> :scheme: http == Indexed - Add == idx = 4 |
| be | -> :path: / == Indexed - Add == idx = 62 |
| 58 | -> :authority: www.example\ .com == Literal indexed == Indexed name (idx = 24) |
| 08 | cache-control |
| 6e6f 2d63 6163 6865 | Literal value (len = 8) 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 86 84 41 8c fle3 c2e5 f23a 6ba0 ab90 f4ff | == Indexed - Add == idx = 2 -> :method: GET == Indexed - Add == idx = 6 -> :scheme: http == Indexed - Add == idx = 4 -> :path: / == Literal indexed == Indexed name (idx = 1) :authority Literal value (len = 12) Huffman encoded::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\sample.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=ASDJKHQKBZXOQWEOPIUAXQWEIOIU; 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=ASDJKHQKBZXOQWEOPIUAXQWEIOIU; 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 | <pre> == Literal indexed == Indexed name (idx = 8) :status </pre> |
| 83 | <pre> Literal value (len = 3) Huffman encoded: </pre> |
| 640e ff | <pre> d.. Decoded: 307 - evict: :status: 302 -> :status: 307 </pre> |
| c1 | <pre> == Indexed - Add == idx = 65 -> cache-control: private </pre> |
| c0 | <pre> == Indexed - Add == idx = 64 -> date: Mon, 21 Oct 2013 \ 20:13:21 GMT </pre> |
| bf | <pre> == Indexed - Add == idx = 63 -> location: https://www.e\ xample.com </pre> |

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: |

| | |
|---|--|
| 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 | ... 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 |
|---|--|

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 an 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.

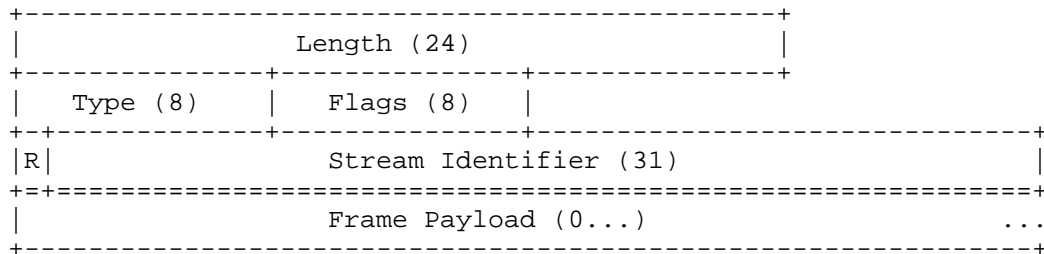


Figure 1: Frame Layout

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

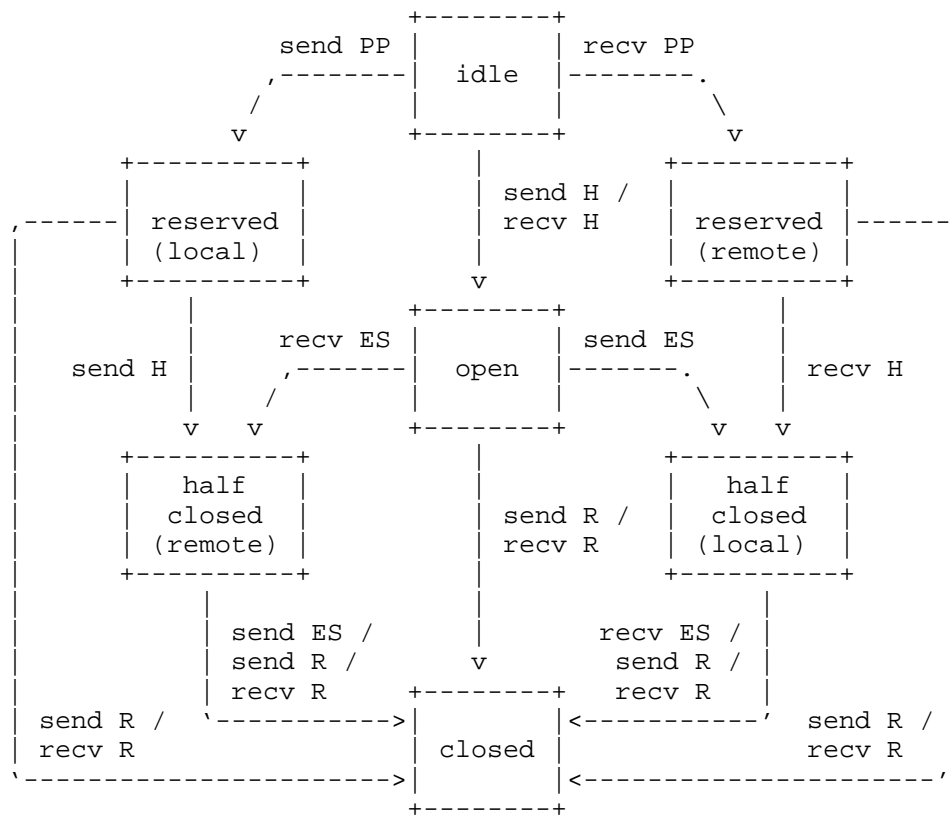
A "stream" is an independent, bi-directional sequence of frames exchanged between the client and server within an HTTP/2 connection. Streams have several important characteristics:

- 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.

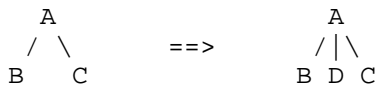


Figure 3: Example of Default Dependency Creation

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.

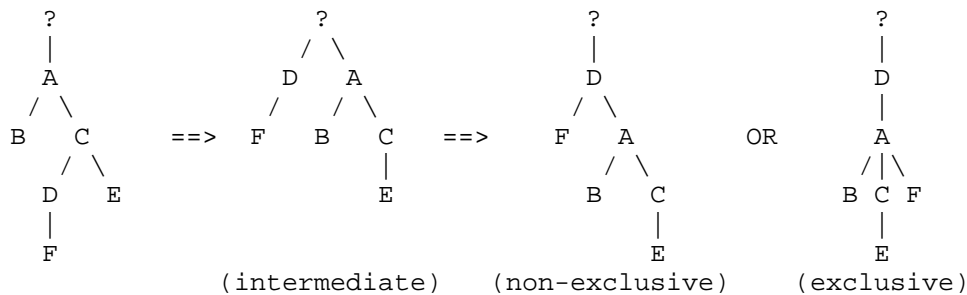


Figure 5: Example of Dependency Reordering

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.

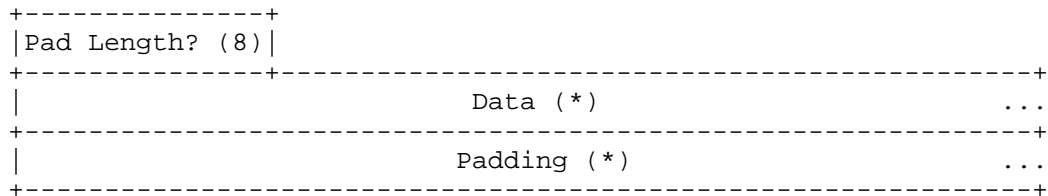


Figure 6: DATA Frame Payload

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.

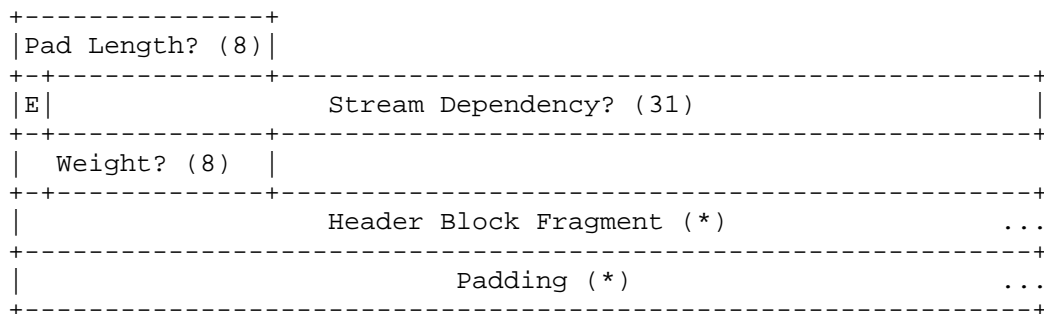


Figure 7: HEADERS Frame Payload

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.

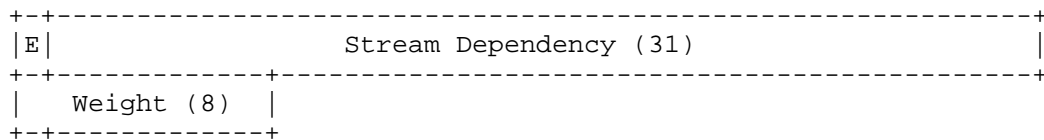


Figure 8: PRIORITY Frame Payload

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.

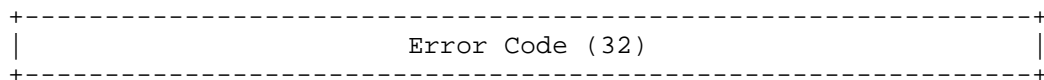


Figure 9: RST_STREAM Frame Payload

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.

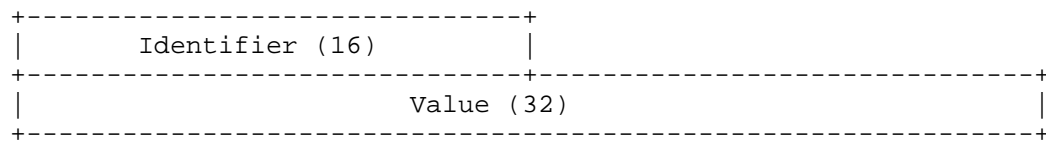


Figure 10: Setting Format

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.

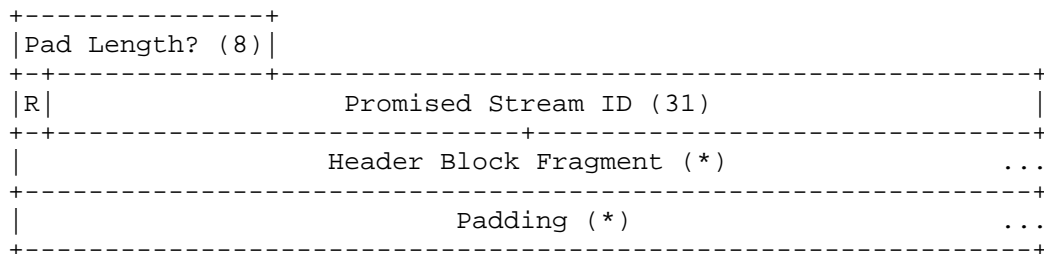


Figure 11: PUSH_PROMISE Payload Format

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.

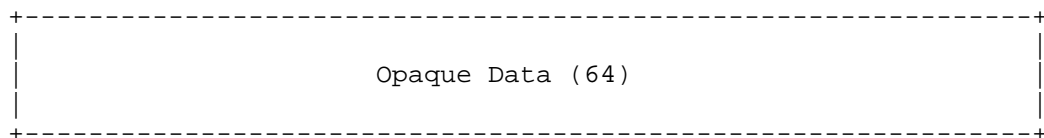


Figure 12: `PING` Payload Format

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.

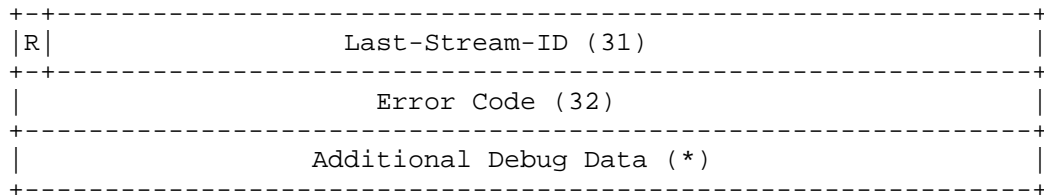


Figure 13: GOAWAY Payload Format

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.

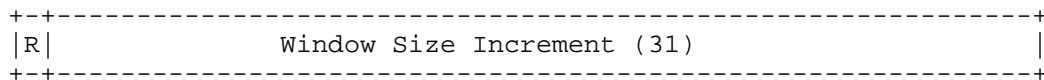


Figure 14: WINDOW_UPDATE Payload Format

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 an 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 | ==> | + <code>END_STREAM</code> |
| Accept: image/jpeg | | + <code>END_HEADERS</code> |
| | | :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" | ==> | + <code>END_STREAM</code> |
| Expires: Thu, 23 Jan ... | | + <code>END_HEADERS</code> |
| | | :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 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

HTTPbis Working Group
Internet-Draft
Intended status: Informational
Expires: August 10, 2014

J. Reschke
greenbytes
February 6, 2014

Initial Hypertext Transfer Protocol (HTTP) Method Registrations
draft-ietf-httpbis-method-registrations-15

Abstract

This document registers those Hypertext Transfer Protocol (HTTP) methods which have been defined in RFCs before the IANA HTTP Method Registry was established.

Editorial Note (To be removed by RFC Editor)

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

The current issues list is at <http://trac.tools.ietf.org/wg/httpbis/trac/query?component=method-registrations> and related documents (including fancy diffs) can be found at <http://tools.ietf.org/wg/httpbis/>.

The changes in this draft are summarized in Appendix A.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 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. Security Considerations | 3 |
| 3. IANA Considerations | 3 |
| 4. Normative References | 4 |
| Appendix A. Change Log (to be removed by RFC Editor before publication) | 6 |
| A.1. Since draft-ietf-httpbis-method-registrations-13 | 6 |
| A.2. Since draft-ietf-httpbis-method-registrations-14 | 6 |

1. Introduction

This document registers those Hypertext Transfer Protocol (HTTP) methods which have been defined in RFCs other than [draft-ietf-httpbis-p2-semantics] before the IANA HTTP Method Registry was established.

2. Security Considerations

There are no security considerations related to the registration itself.

Security considerations applicable to the individual HTTP methods ought to be discussed in the specifications that define them.

3. IANA Considerations

The table below provides registrations of HTTP method names to be added to the IANA HTTP Method registry at <http://www.iana.org/assignments/http-methods> (see Section 8.1 of [draft-ietf-httpbis-p2-semantics]).

| Method Name | Safe | Idempotent | Reference |
|-------------------|------|------------|--|
| ACL | no | yes | [RFC3744], Section 8.1 |
| BASELINE-CONTROL | no | yes | [RFC3253], Section 12.6 |
| BIND | no | yes | [RFC5842], Section 4 |
| CHECKIN | no | yes | [RFC3253], Section 4.4 and [RFC3253], Section 9.4 |
| CHECKOUT | no | yes | [RFC3253], Section 4.3 and [RFC3253], Section 8.8 |
| COPY | no | yes | [RFC4918], Section 9.8 |
| LABEL | no | yes | [RFC3253], Section 8.2 |
| LINK | no | yes | [RFC2068], Section 19.6.1.2 |
| LOCK | no | no | [RFC4918], Section 9.10 |
| MERGE | no | yes | [RFC3253], Section 11.2 |
| MKACTIVITY | no | yes | [RFC3253], Section 13.5 |
| MKCALENDAR | no | yes | [RFC4791], Section 5.3.1 |
| MKCOL | no | yes | [RFC4918], Section 9.3 |
| MKREDIRECTREF | no | yes | [RFC4437], Section 6 |
| MKWORKSPACE | no | yes | [RFC3253], Section 6.3 |
| MOVE | no | yes | [RFC4918], Section 9.9 |
| ORDERPATCH | no | yes | [RFC3648], Section 7 |
| PATCH | no | no | [RFC5789], Section 2 |
| PROPFIND | yes | yes | [RFC4918], Section 9.1 |
| PROPPATCH | no | yes | [RFC4918], Section 9.2 |
| REBIND | no | yes | [RFC5842], Section 6 |
| REPORT | yes | yes | [RFC3253], Section 3.6 |
| SEARCH | yes | yes | [RFC5323], Section 2 |
| UNBIND | no | yes | [RFC5842], Section 5 |
| UNCHECKOUT | no | yes | [RFC3253], Section 4.5 |
| UNLINK | no | yes | [RFC2068], Section 19.6.1.3 |
| UNLOCK | no | yes | [RFC4918], Section 9.11 |
| UPDATE | no | yes | [RFC3253], Section 7.1 |
| UPDATEREDIRECTREF | no | yes | [RFC4437], Section 7 |
| VERSION-CONTROL | no | yes | [RFC3253], Section 3.5 |

4. Normative References

- [RFC2068] Fielding, R., Gettys, J., Mogul, J., Nielsen, H., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2068, January 1997.

- [RFC3253] Clemm, G., Amsden, J., Ellison, T., Kaler, C., and J. Whitehead, "Versioning Extensions to WebDAV (Web Distributed Authoring and Versioning)", RFC 3253, March 2002.
- [RFC3648] Whitehead, J. and J. Reschke, Ed., "Web Distributed Authoring and Versioning (WebDAV) Ordered Collections Protocol", RFC 3648, December 2003.
- [RFC3744] Clemm, G., Reschke, J., Sedlar, E., and J. Whitehead, "Web Distributed Authoring and Versioning (WebDAV) Access Control Protocol", RFC 3744, May 2004.
- [RFC4437] Whitehead, J., Clemm, G., and J. Reschke, Ed., "Web Distributed Authoring and Versioning (WebDAV) Redirect Reference Resources", RFC 4437, March 2006.
- [RFC4791] Daboo, C., Desruisseaux, B., and L. Dusseault, "Calendaring Extensions to WebDAV (CalDAV)", RFC 4791, March 2007.
- [RFC4918] Dusseault, L., Ed., "HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV)", RFC 4918, June 2007.
- [RFC5323] Reschke, J., Ed., Reddy, S., Davis, J., and A. Babich, "Web Distributed Authoring and Versioning (WebDAV) SEARCH", RFC 5323, November 2008.
- [RFC5789] Dusseault, L. and J. Snell, "PATCH Method for HTTP", RFC 5789, March 2010.
- [RFC5842] Clemm, G., Crawford, J., Reschke, J., Ed., and J. Whitehead, "Binding Extensions to Web

Distributed Authoring and
Versioning (WebDAV)", RFC 5842,
April 2010.

[draft-ietf-httpbis-p2-semantics] Fielding, R., Ed. and J. Reschke,
Ed., "Hypertext Transfer Protocol
(HTTP/1.1): Semantics and
Content",
draft-ietf-httpbis-p2-semantics-26
(work in progress), February 2014.

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

Changes up to the IETF Last Call draft are summarized in <http://trac.tools.ietf.org/html/draft-ietf-httpbis-method-registrations-13#appendix-B>.

A.1. Since draft-ietf-httpbis-method-registrations-13

Closed issues:

- o <http://tools.ietf.org/wg/httpbis/trac/ticket/514>: "registration tables should be inside IANA considerations"

Clarified the IANA action to say "add".

Updated httpbis reference.

A.2. Since draft-ietf-httpbis-method-registrations-14

Closed issues:

- o <http://tools.ietf.org/wg/httpbis/trac/ticket/529>: "IESG ballot on draft-ietf-httpbis-method-registrations-14"

Removed misleading statement about "standards-track" RFCs, as some of the methods registered here indeed originate from Experimental RFCs, and furthermore the new registry established in Section 8.1 of [draft-ietf-httpbis-p2-semantics] uses "IETF Review".

Updated httpbis reference.

Author's Address

Julian F. Reschke
greenbytes GmbH
Hafenweg 16
Muenster, NW 48155
Germany

EMail: julian.reschke@greenbytes.de
URI: <http://greenbytes.de/tech/webdav/>

HTTPbis Working Group
Internet-Draft
Obsoletes: 2145,2616 (if approved)
Updates: 2817,2818 (if approved)
Intended status: Standards Track
Expires: August 10, 2014

R. Fielding, Ed.
Adobe
J. Reschke, Ed.
greenbytes
February 6, 2014

Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing
draft-ietf-httpbis-pl-messaging-26

Abstract

The Hypertext Transfer Protocol (HTTP) is a stateless application-level protocol for distributed, collaborative, hypertext information systems. This document provides an overview of HTTP architecture and its associated terminology, defines the "http" and "https" Uniform Resource Identifier (URI) schemes, defines the HTTP/1.1 message syntax and parsing requirements, and describes related security concerns for implementations.

Editorial Note (To be removed by RFC Editor)

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

The current issues list is at <http://tools.ietf.org/wg/httpbis/trac/report/3> and related documents (including fancy diffs) can be found at <http://tools.ietf.org/wg/httpbis/>.

The changes in this draft are summarized in Appendix C.3.

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 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.

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.

Table of Contents

| | |
|--|----|
| 1. Introduction | 5 |
| 1.1. Requirement Notation | 6 |
| 1.2. Syntax Notation | 6 |
| 2. Architecture | 6 |
| 2.1. Client/Server Messaging | 7 |
| 2.2. Implementation Diversity | 8 |
| 2.3. Intermediaries | 9 |
| 2.4. Caches | 11 |
| 2.5. Conformance and Error Handling | 12 |
| 2.6. Protocol Versioning | 13 |
| 2.7. Uniform Resource Identifiers | 16 |
| 2.7.1. http URI scheme | 16 |
| 2.7.2. https URI scheme | 18 |
| 2.7.3. http and https URI Normalization and Comparison | 19 |
| 3. Message Format | 19 |
| 3.1. Start Line | 20 |
| 3.1.1. Request Line | 21 |

| | | |
|--------|---|----|
| 3.1.2. | Status Line | 22 |
| 3.2. | Header Fields | 22 |
| 3.2.1. | Field Extensibility | 23 |
| 3.2.2. | Field Order | 23 |
| 3.2.3. | Whitespace | 24 |
| 3.2.4. | Field Parsing | 25 |
| 3.2.5. | Field Limits | 26 |
| 3.2.6. | Field value components | 26 |
| 3.3. | Message Body | 27 |
| 3.3.1. | Transfer-Encoding | 28 |
| 3.3.2. | Content-Length | 30 |
| 3.3.3. | Message Body Length | 31 |
| 3.4. | Handling Incomplete Messages | 33 |
| 3.5. | Message Parsing Robustness | 34 |
| 4. | Transfer Codings | 35 |
| 4.1. | Chunked Transfer Coding | 35 |
| 4.1.1. | Chunk Extensions | 36 |
| 4.1.2. | Chunked Trailer Part | 36 |
| 4.1.3. | Decoding Chunked | 37 |
| 4.2. | Compression Codings | 38 |
| 4.2.1. | Compress Coding | 38 |
| 4.2.2. | Deflate Coding | 38 |
| 4.2.3. | Gzip Coding | 38 |
| 4.3. | TE | 38 |
| 4.4. | Trailer | 39 |
| 5. | Message Routing | 40 |
| 5.1. | Identifying a Target Resource | 40 |
| 5.2. | Connecting Inbound | 40 |
| 5.3. | Request Target | 41 |
| 5.3.1. | origin-form | 41 |
| 5.3.2. | absolute-form | 42 |
| 5.3.3. | authority-form | 42 |
| 5.3.4. | asterisk-form | 42 |
| 5.4. | Host | 43 |
| 5.5. | Effective Request URI | 44 |
| 5.6. | Associating a Response to a Request | 46 |
| 5.7. | Message Forwarding | 46 |
| 5.7.1. | Via | 47 |
| 5.7.2. | Transformations | 48 |
| 6. | Connection Management | 49 |
| 6.1. | Connection | 50 |
| 6.2. | Establishment | 51 |
| 6.3. | Persistence | 52 |
| 6.3.1. | Retrying Requests | 53 |
| 6.3.2. | Pipelining | 53 |
| 6.4. | Concurrency | 54 |
| 6.5. | Failures and Time-outs | 54 |
| 6.6. | Tear-down | 55 |

| | |
|--|----|
| 6.7. Upgrade | 56 |
| 7. ABNF list extension: #rule | 58 |
| 8. IANA Considerations | 60 |
| 8.1. Header Field Registration | 60 |
| 8.2. URI Scheme Registration | 60 |
| 8.3. Internet Media Type Registration | 61 |
| 8.3.1. Internet Media Type message/http | 61 |
| 8.3.2. Internet Media Type application/http | 62 |
| 8.4. Transfer Coding Registry | 63 |
| 8.4.1. Procedure | 63 |
| 8.4.2. Registration | 64 |
| 8.5. Content Coding Registration | 64 |
| 8.6. Upgrade Token Registry | 65 |
| 8.6.1. Procedure | 65 |
| 8.6.2. Upgrade Token Registration | 66 |
| 9. Security Considerations | 66 |
| 9.1. Establishing Authority | 66 |
| 9.2. Risks of Intermediaries | 67 |
| 9.3. Attacks via Protocol Element Length | 68 |
| 9.4. Response Splitting | 68 |
| 9.5. Request Smuggling | 69 |
| 9.6. Message Integrity | 69 |
| 9.7. Message Confidentiality | 70 |
| 9.8. Privacy of Server Log Information | 70 |
| 10. Acknowledgments | 71 |
| 11. References | 72 |
| 11.1. Normative References | 72 |
| 11.2. Informative References | 74 |
| Appendix A. HTTP Version History | 76 |
| A.1. Changes from HTTP/1.0 | 77 |
| A.1.1. Multi-homed Web Servers | 77 |
| A.1.2. Keep-Alive Connections | 77 |
| A.1.3. Introduction of Transfer-Encoding | 78 |
| A.2. Changes from RFC 2616 | 78 |
| Appendix B. Collected ABNF | 80 |
| Appendix C. Change Log (to be removed by RFC Editor before publication) | 82 |
| C.1. Since RFC 2616 | 82 |
| C.2. Since draft-ietf-httpbis-pl-messaging-24 | 83 |
| C.3. Since draft-ietf-httpbis-pl-messaging-25 | 83 |
| Index | 84 |

1. Introduction

The Hypertext Transfer Protocol (HTTP) is a stateless application-level request/response protocol that uses extensible semantics and self-descriptive message payloads for flexible interaction with network-based hypertext information systems. This document is the first in a series of documents that collectively form the HTTP/1.1 specification:

RFC xxx1: Message Syntax and Routing

RFC xxx2: Semantics and Content

RFC xxx3: Conditional Requests

RFC xxx4: Range Requests

RFC xxx5: Caching

RFC xxx6: Authentication

This HTTP/1.1 specification obsoletes RFC 2616 and RFC 2145 (on HTTP versioning). This specification also updates the use of CONNECT to establish a tunnel, previously defined in RFC 2817, and defines the "https" URI scheme that was described informally in RFC 2818.

HTTP is a generic interface protocol for information systems. It is designed to hide the details of how a service is implemented by presenting a uniform interface to clients that is independent of the types of resources provided. Likewise, servers do not need to be aware of each client's purpose: an HTTP request can be considered in isolation rather than being associated with a specific type of client or a predetermined sequence of application steps. The result is a protocol that can be used effectively in many different contexts and for which implementations can evolve independently over time.

HTTP is also designed for use as an intermediation protocol for translating communication to and from non-HTTP information systems. HTTP proxies and gateways can provide access to alternative information services by translating their diverse protocols into a hypertext format that can be viewed and manipulated by clients in the same way as HTTP services.

One consequence of this flexibility is that the protocol cannot be defined in terms of what occurs behind the interface. Instead, we are limited to defining the syntax of communication, the intent of received communication, and the expected behavior of recipients. If the communication is considered in isolation, then successful actions

ought to be reflected in corresponding changes to the observable interface provided by servers. However, since multiple clients might act in parallel and perhaps at cross-purposes, we cannot require that such changes be observable beyond the scope of a single response.

This document describes the architectural elements that are used or referred to in HTTP, defines the "http" and "https" URI schemes, describes overall network operation and connection management, and defines HTTP message framing and forwarding requirements. Our goal is to define all of the mechanisms necessary for HTTP message handling that are independent of message semantics, thereby defining the complete set of requirements for message parsers and message-forwarding intermediaries.

1.1. Requirement Notation

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

Conformance criteria and considerations regarding error handling are defined in Section 2.5.

1.2. Syntax Notation

This specification uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234] with a list extension, defined in Section 7, that allows for compact definition of comma-separated lists using a '#' operator (similar to how the '*' operator indicates repetition). Appendix B shows the collected grammar with all list operators expanded to standard ABNF notation.

The following core rules are included by reference, as defined in [RFC5234], Appendix B.1: ALPHA (letters), CR (carriage return), CRLF (CR LF), CTL (controls), DIGIT (decimal 0-9), DQUOTE (double quote), HEXDIG (hexadecimal 0-9/A-F/a-f), HTAB (horizontal tab), LF (line feed), OCTET (any 8-bit sequence of data), SP (space), and VCHAR (any visible [USASCII] character).

As a convention, ABNF rule names prefixed with "obs-" denote "obsolete" grammar rules that appear for historical reasons.

2. Architecture

HTTP was created for the World Wide Web (WWW) architecture and has evolved over time to support the scalability needs of a worldwide hypertext system. Much of that architecture is reflected in the terminology and syntax productions used to define HTTP.

2.1. Client/Server Messaging

HTTP is a stateless request/response protocol that operates by exchanging messages (Section 3) across a reliable transport or session-layer "connection" (Section 6). An HTTP "client" is a program that establishes a connection to a server for the purpose of sending one or more HTTP requests. An HTTP "server" is a program that accepts connections in order to service HTTP requests by sending HTTP responses.

The terms client and server refer only to the roles that these programs perform for a particular connection. The same program might act as a client on some connections and a server on others. The term "user agent" refers to any of the various client programs that initiate a request, including (but not limited to) browsers, spiders (web-based robots), command-line tools, custom applications, and mobile apps. The term "origin server" refers to the program that can originate authoritative responses for a given target resource. The terms "sender" and "recipient" refer to any implementation that sends or receives a given message, respectively.

HTTP relies upon the Uniform Resource Identifier (URI) standard [RFC3986] to indicate the target resource (Section 5.1) and relationships between resources. Messages are passed in a format similar to that used by Internet mail [RFC5322] and the Multipurpose Internet Mail Extensions (MIME) [RFC2045] (see Appendix A of [Part2] for the differences between HTTP and MIME messages).

Most HTTP communication consists of a retrieval request (GET) for a representation of some resource identified by a URI. In the simplest case, this might be accomplished via a single bidirectional connection (==) between the user agent (UA) and the origin server (O).

```
request    >
UA ===== O
              < response
```

A client sends an HTTP request to a server in the form of a request message, beginning with a request-line that includes a method, URI, and protocol version (Section 3.1.1), followed by header fields containing request modifiers, client information, and representation metadata (Section 3.2), an empty line to indicate the end of the header section, and finally a message body containing the payload body (if any, Section 3.3).

A server responds to a client's request by sending one or more HTTP response messages, each beginning with a status line that includes

the protocol version, a success or error code, and textual reason phrase (Section 3.1.2), possibly followed by header fields containing server information, resource metadata, and representation metadata (Section 3.2), an empty line to indicate the end of the header section, and finally a message body containing the payload body (if any, Section 3.3).

A connection might be used for multiple request/response exchanges, as defined in Section 6.3.

The following example illustrates a typical message exchange for a GET request (Section 4.3.1 of [Part2]) on the URI "http://www.example.com/hello.txt":

Client request:

```
GET /hello.txt HTTP/1.1
User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.7l zlib/1.2.3
Host: www.example.com
Accept-Language: en, mi
```

Server response:

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
ETag: "34aa387-d-1568eb00"
Accept-Ranges: bytes
Content-Length: 51
Vary: Accept-Encoding
Content-Type: text/plain
```

Hello World! My payload includes a trailing CRLF.

2.2. Implementation Diversity

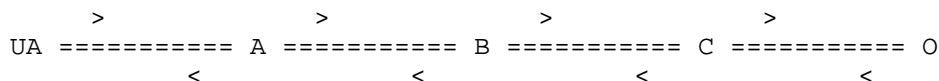
When considering the design of HTTP, it is easy to fall into a trap of thinking that all user agents are general-purpose browsers and all origin servers are large public websites. That is not the case in practice. Common HTTP user agents include household appliances, stereos, scales, firmware update scripts, command-line programs, mobile apps, and communication devices in a multitude of shapes and sizes. Likewise, common HTTP origin servers include home automation units, configurable networking components, office machines, autonomous robots, news feeds, traffic cameras, ad selectors, and video delivery platforms.

The term "user agent" does not imply that there is a human user directly interacting with the software agent at the time of a request. In many cases, a user agent is installed or configured to run in the background and save its results for later inspection (or save only a subset of those results that might be interesting or erroneous). Spiders, for example, are typically given a start URI and configured to follow certain behavior while crawling the Web as a hypertext graph.

The implementation diversity of HTTP means that not all user agents can make interactive suggestions to their user or provide adequate warning for security or privacy concerns. In the few cases where this specification requires reporting of errors to the user, it is acceptable for such reporting to only be observable in an error console or log file. Likewise, requirements that an automated action be confirmed by the user before proceeding might be met via advance configuration choices, run-time options, or simple avoidance of the unsafe action; confirmation does not imply any specific user interface or interruption of normal processing if the user has already made that choice.

2.3. Intermediaries

HTTP enables the use of intermediaries to satisfy requests through a chain of connections. There are three common forms of HTTP intermediary: proxy, gateway, and tunnel. In some cases, a single intermediary might act as an origin server, proxy, gateway, or tunnel, switching behavior based on the nature of each request.



The figure above shows three intermediaries (A, B, and C) between the user agent and origin server. A request or response message that travels the whole chain will pass through four separate connections. Some HTTP communication options might apply only to the connection with the nearest, non-tunnel neighbor, only to the end-points of the chain, or to all connections along the chain. Although the diagram is linear, each participant might be engaged in multiple, simultaneous communications. For example, B might be receiving requests from many clients other than A, and/or forwarding requests to servers other than C, at the same time that it is handling A's request. Likewise, later requests might be sent through a different path of connections, often based on dynamic configuration for load balancing.

The terms "upstream" and "downstream" are used to describe

directional requirements in relation to the message flow: all messages flow from upstream to downstream. The terms inbound and outbound are used to describe directional requirements in relation to the request route: "inbound" means toward the origin server and "outbound" means toward the user agent.

A "proxy" is a message forwarding agent that is selected by the client, usually via local configuration rules, to receive requests for some type(s) of absolute URI and attempt to satisfy those requests via translation through the HTTP interface. Some translations are minimal, such as for proxy requests for "http" URIs, whereas other requests might require translation to and from entirely different application-level protocols. Proxies are often used to group an organization's HTTP requests through a common intermediary for the sake of security, annotation services, or shared caching. Some proxies are designed to apply transformations to selected messages or payloads while they are being forwarded, as described in Section 5.7.2.

A "gateway" (a.k.a., "reverse proxy") is an intermediary that acts as an origin server for the outbound connection, but translates received requests and forwards them inbound to another server or servers. Gateways are often used to encapsulate legacy or untrusted information services, to improve server performance through "accelerator" caching, and to enable partitioning or load balancing of HTTP services across multiple machines.

All HTTP requirements applicable to an origin server also apply to the outbound communication of a gateway. A gateway communicates with inbound servers using any protocol that it desires, including private extensions to HTTP that are outside the scope of this specification. However, an HTTP-to-HTTP gateway that wishes to interoperate with third-party HTTP servers ought to conform to user agent requirements on the gateway's inbound connection.

A "tunnel" acts as a blind relay between two connections without changing the messages. Once active, a tunnel is not considered a party to the HTTP communication, though the tunnel might have been initiated by an HTTP request. A tunnel ceases to exist when both ends of the relayed connection are closed. Tunnels are used to extend a virtual connection through an intermediary, such as when Transport Layer Security (TLS, [RFC5246]) is used to establish confidential communication through a shared firewall proxy.

The above categories for intermediary only consider those acting as participants in the HTTP communication. There are also intermediaries that can act on lower layers of the network protocol stack, filtering or redirecting HTTP traffic without the knowledge or

permission of message senders. Network intermediaries are indistinguishable (at a protocol level) from a man-in-the-middle attack, often introducing security flaws or interoperability problems due to mistakenly violating HTTP semantics.

For example, an "interception proxy" [RFC3040] (also commonly known as a "transparent proxy" [RFC1919] or "captive portal") differs from an HTTP proxy because it is not selected by the client. Instead, an interception proxy filters or redirects outgoing TCP port 80 packets (and occasionally other common port traffic). Interception proxies are commonly found on public network access points, as a means of enforcing account subscription prior to allowing use of non-local Internet services, and within corporate firewalls to enforce network usage policies.

HTTP is defined as a stateless protocol, meaning that each request message can be understood in isolation. Many implementations depend on HTTP's stateless design in order to reuse proxied connections or dynamically load-balance requests across multiple servers. Hence, a server **MUST NOT** assume that two requests on the same connection are from the same user agent unless the connection is secured and specific to that agent. Some non-standard HTTP extensions (e.g., [RFC4559]) have been known to violate this requirement, resulting in security and interoperability problems.

2.4. Caches

A "cache" is a local store of previous response messages and the subsystem that controls its message storage, retrieval, and deletion. A cache stores cacheable responses in order to reduce the response time and network bandwidth consumption on future, equivalent requests. Any client or server **MAY** employ a cache, though a cache cannot be used by a server while it is acting as a tunnel.

The effect of a cache is that the request/response chain is shortened if one of the participants along the chain has a cached response applicable to that request. The following illustrates the resulting chain if B has a cached copy of an earlier response from O (via C) for a request that has not been cached by UA or A.

```

      >               >
UA ===== A ===== B - - - - - C - - - - - O
      <               <

```

A response is "cacheable" if a cache is allowed to store a copy of the response message for use in answering subsequent requests. Even when a response is cacheable, there might be additional constraints placed by the client or by the origin server on when that cached

response can be used for a particular request. HTTP requirements for cache behavior and cacheable responses are defined in Section 2 of [Part6].

There are a wide variety of architectures and configurations of caches deployed across the World Wide Web and inside large organizations. These include national hierarchies of proxy caches to save transoceanic bandwidth, collaborative systems that broadcast or multicast cache entries, archives of pre-fetched cache entries for use in off-line or high-latency environments, and so on.

2.5. Conformance and Error Handling

This specification targets conformance criteria according to the role of a participant in HTTP communication. Hence, HTTP requirements are placed on senders, recipients, clients, servers, user agents, intermediaries, origin servers, proxies, gateways, or caches, depending on what behavior is being constrained by the requirement. Additional (social) requirements are placed on implementations, resource owners, and protocol element registrations when they apply beyond the scope of a single communication.

The verb "generate" is used instead of "send" where a requirement differentiates between creating a protocol element and merely forwarding a received element downstream.

An implementation is considered conformant if it complies with all of the requirements associated with the roles it partakes in HTTP.

Conformance includes both the syntax and semantics of protocol elements. A sender **MUST NOT** generate protocol elements that convey a meaning that is known by that sender to be false. A sender **MUST NOT** generate protocol elements that do not match the grammar defined by the corresponding ABNF rules. Within a given message, a sender **MUST NOT** generate protocol elements or syntax alternatives that are only allowed to be generated by participants in other roles (i.e., a role that the sender does not have for that message).

When a received protocol element is parsed, the recipient **MUST** be able to parse any value of reasonable length that is applicable to the recipient's role and matches the grammar defined by the corresponding ABNF rules. Note, however, that some received protocol elements might not be parsed. For example, an intermediary forwarding a message might parse a header-field into generic field-name and field-value components, but then forward the header field without further parsing inside the field-value.

HTTP does not have specific length limitations for many of its

protocol elements because the lengths that might be appropriate will vary widely, depending on the deployment context and purpose of the implementation. Hence, interoperability between senders and recipients depends on shared expectations regarding what is a reasonable length for each protocol element. Furthermore, what is commonly understood to be a reasonable length for some protocol elements has changed over the course of the past two decades of HTTP use, and is expected to continue changing in the future.

At a minimum, a recipient **MUST** be able to parse and process protocol element lengths that are at least as long as the values that it generates for those same protocol elements in other messages. For example, an origin server that publishes very long URI references to its own resources needs to be able to parse and process those same references when received as a request target.

A recipient **MUST** interpret a received protocol element according to the semantics defined for it by this specification, including extensions to this specification, unless the recipient has determined (through experience or configuration) that the sender incorrectly implements what is implied by those semantics. For example, an origin server might disregard the contents of a received Accept-Encoding header field if inspection of the User-Agent header field indicates a specific implementation version that is known to fail on receipt of certain content codings.

Unless noted otherwise, a recipient **MAY** attempt to recover a usable protocol element from an invalid construct. HTTP does not define specific error handling mechanisms except when they have a direct impact on security, since different applications of the protocol require different error handling strategies. For example, a Web browser might wish to transparently recover from a response where the Location header field doesn't parse according to the ABNF, whereas a systems control client might consider any form of error recovery to be dangerous.

2.6. Protocol Versioning

HTTP uses a "<major>.<minor>" numbering scheme to indicate versions of the protocol. This specification defines version "1.1". The protocol version as a whole indicates the sender's conformance with the set of requirements laid out in that version's corresponding specification of HTTP.

The version of an HTTP message is indicated by an HTTP-version field in the first line of the message. HTTP-version is case-sensitive.

HTTP-version = HTTP-name "/" DIGIT "." DIGIT

HTTP-name = %x48.54.54.50 ; "HTTP", case-sensitive

The HTTP version number consists of two decimal digits separated by a "." (period or decimal point). The first digit ("major version") indicates the HTTP messaging syntax, whereas the second digit ("minor version") indicates the highest minor version within that major version to which the sender is conformant and able to understand for future communication. The minor version advertises the sender's communication capabilities even when the sender is only using a backwards-compatible subset of the protocol, thereby letting the recipient know that more advanced features can be used in response (by servers) or in future requests (by clients).

When an HTTP/1.1 message is sent to an HTTP/1.0 recipient [RFC1945] or a recipient whose version is unknown, the HTTP/1.1 message is constructed such that it can be interpreted as a valid HTTP/1.0 message if all of the newer features are ignored. This specification places recipient-version requirements on some new features so that a conformant sender will only use compatible features until it has determined, through configuration or the receipt of a message, that the recipient supports HTTP/1.1.

The interpretation of a header field does not change between minor versions of the same major HTTP version, though the default behavior of a recipient in the absence of such a field can change. Unless specified otherwise, header fields defined in HTTP/1.1 are defined for all versions of HTTP/1.x. In particular, the Host and Connection header fields ought to be implemented by all HTTP/1.x implementations whether or not they advertise conformance with HTTP/1.1.

New header fields can be introduced without changing the protocol version if their defined semantics allow them to be safely ignored by recipients that do not recognize them. Header field extensibility is discussed in Section 3.2.1.

Intermediaries that process HTTP messages (i.e., all intermediaries other than those acting as tunnels) MUST send their own HTTP-version in forwarded messages. In other words, they are not allowed to blindly forward the first line of an HTTP message without ensuring that the protocol version in that message matches a version to which that intermediary is conformant for both the receiving and sending of messages. Forwarding an HTTP message without rewriting the HTTP-version might result in communication errors when downstream recipients use the message sender's version to determine what features are safe to use for later communication with that sender.

A client SHOULD send a request version equal to the highest version to which the client is conformant and whose major version is no

higher than the highest version supported by the server, if this is known. A client MUST NOT send a version to which it is not conformant.

A client MAY send a lower request version if it is known that the server incorrectly implements the HTTP specification, but only after the client has attempted at least one normal request and determined from the response status code or header fields (e.g., Server) that the server improperly handles higher request versions.

A server SHOULD send a response version equal to the highest version to which the server is conformant that has a major version less than or equal to the one received in the request. A server MUST NOT send a version to which it is not conformant. A server can send a 505 (HTTP Version Not Supported) response if it wishes, for any reason, to refuse service of the client's major protocol version.

A server MAY send an HTTP/1.0 response to a request if it is known or suspected that the client incorrectly implements the HTTP specification and is incapable of correctly processing later version responses, such as when a client fails to parse the version number correctly or when an intermediary is known to blindly forward the HTTP-version even when it doesn't conform to the given minor version of the protocol. Such protocol downgrades SHOULD NOT be performed unless triggered by specific client attributes, such as when one or more of the request header fields (e.g., User-Agent) uniquely match the values sent by a client known to be in error.

The intention of HTTP's versioning design is that the major number will only be incremented if an incompatible message syntax is introduced, and that the minor number will only be incremented when changes made to the protocol have the effect of adding to the message semantics or implying additional capabilities of the sender. However, the minor version was not incremented for the changes introduced between [RFC2068] and [RFC2616], and this revision has specifically avoided any such changes to the protocol.

When an HTTP message is received with a major version number that the recipient implements, but a higher minor version number than what the recipient implements, the recipient SHOULD process the message as if it were in the highest minor version within that major version to which the recipient is conformant. A recipient can assume that a message with a higher minor version, when sent to a recipient that has not yet indicated support for that higher version, is sufficiently backwards-compatible to be safely processed by any implementation of the same major version.

2.7. Uniform Resource Identifiers

Uniform Resource Identifiers (URIs) [RFC3986] are used throughout HTTP as the means for identifying resources (Section 2 of [Part2]). URI references are used to target requests, indicate redirects, and define relationships.

The definitions of "URI-reference", "absolute-URI", "relative-part", "scheme", "authority", "port", "host", "path-abempty", "segment", "query", and "fragment" are adopted from the URI generic syntax. An "absolute-path" rule is defined for protocol elements that can contain a non-empty path component. (This rule differs slightly from RFC 3986's path-abempty rule, which allows for an empty path to be used in references, and path-absolute rule, which does not allow paths that begin with "//".) A "partial-URI" rule is defined for protocol elements that can contain a relative URI but not a fragment component.

```
URI-reference = <URI-reference, defined in [RFC3986], Section 4.1>
absolute-URI  = <absolute-URI, defined in [RFC3986], Section 4.3>
relative-part = <relative-part, defined in [RFC3986], Section 4.2>
scheme        = <scheme, defined in [RFC3986], Section 3.1>
authority     = <authority, defined in [RFC3986], Section 3.2>
uri-host      = <host, defined in [RFC3986], Section 3.2.2>
port          = <port, defined in [RFC3986], Section 3.2.3>
path-abempty  = <path-abempty, defined in [RFC3986], Section 3.3>
segment       = <segment, defined in [RFC3986], Section 3.3>
query         = <query, defined in [RFC3986], Section 3.4>
fragment      = <fragment, defined in [RFC3986], Section 3.5>

absolute-path = 1*( "/" segment )
partial-URI   = relative-part [ "?" query ]
```

Each protocol element in HTTP that allows a URI reference will indicate in its ABNF production whether the element allows any form of reference (URI-reference), only a URI in absolute form (absolute-URI), only the path and optional query components, or some combination of the above. Unless otherwise indicated, URI references are parsed relative to the effective request URI (Section 5.5).

2.7.1. http URI scheme

The "http" URI scheme is hereby defined for the purpose of minting identifiers according to their association with the hierarchical namespace governed by a potential HTTP origin server listening for TCP ([RFC0793]) connections on a given port.

```
http-URI = "http:" "/" authority path-abempty [ "?" query ]
```

["#" fragment]

The origin server for an "http" URI is identified by the authority component, which includes a host identifier and optional TCP port ([RFC3986], Section 3.2.2). The hierarchical path component and optional query component serve as an identifier for a potential target resource within that origin server's name space. The optional fragment component allows for indirect identification of a secondary resource, independent of the URI scheme, as defined in Section 3.5 of [RFC3986].

A sender **MUST NOT** generate an "http" URI with an empty host identifier. A recipient that processes such a URI reference **MUST** reject it as invalid.

If the host identifier is provided as an IP address, the origin server is the listener (if any) on the indicated TCP port at that IP address. If host is a registered name, the registered name is an indirect identifier for use with a name resolution service, such as DNS, to find an address for that origin server. If the port subcomponent is empty or not given, TCP port 80 (the reserved port for WWW services) is the default.

Note that the presence of a URI with a given authority component does not imply that there is always an HTTP server listening for connections on that host and port. Anyone can mint a URI. What the authority component determines is who has the right to respond authoritatively to requests that target the identified resource. The delegated nature of registered names and IP addresses creates a federated namespace, based on control over the indicated host and port, whether or not an HTTP server is present. See Section 9.1 for security considerations related to establishing authority.

When an "http" URI is used within a context that calls for access to the indicated resource, a client **MAY** attempt access by resolving the host to an IP address, establishing a TCP connection to that address on the indicated port, and sending an HTTP request message (Section 3) containing the URI's identifying data (Section 5) to the server. If the server responds to that request with a non-interim HTTP response message, as described in Section 6 of [Part2], then that response is considered an authoritative answer to the client's request.

Although HTTP is independent of the transport protocol, the "http" scheme is specific to TCP-based services because the name delegation process depends on TCP for establishing authority. An HTTP service based on some other underlying connection protocol would presumably be identified using a different URI scheme, just as the "https"

scheme (below) is used for resources that require an end-to-end secured connection. Other protocols might also be used to provide access to "http" identified resources -- it is only the authoritative interface that is specific to TCP.

The URI generic syntax for authority also includes a deprecated userinfo subcomponent ([RFC3986], Section 3.2.1) for including user authentication information in the URI. Some implementations make use of the userinfo component for internal configuration of authentication information, such as within command invocation options, configuration files, or bookmark lists, even though such usage might expose a user identifier or password. A sender MUST NOT generate the userinfo subcomponent (and its "@" delimiter) when an "http" URI reference is generated within a message as a request target or header field value. Before making use of an "http" URI reference received from an untrusted source, a recipient SHOULD parse for userinfo and treat its presence as an error; it is likely being used to obscure the authority for the sake of phishing attacks.

2.7.2. https URI scheme

The "https" URI scheme is hereby defined for the purpose of minting identifiers according to their association with the hierarchical namespace governed by a potential HTTP origin server listening to a given TCP port for TLS-secured connections ([RFC5246]).

All of the requirements listed above for the "http" scheme are also requirements for the "https" scheme, except that TCP port 443 is the default if the port subcomponent is empty or not given, and the user agent MUST ensure that its connection to the origin server is secured through the use of strong encryption, end-to-end, prior to sending the first HTTP request.

```
https-URI = "https:" "://" authority path-abempty [ "?" query ]
           [ "#" fragment ]
```

Note that the "https" URI scheme depends on both TLS and TCP for establishing authority. Resources made available via the "https" scheme have no shared identity with the "http" scheme even if their resource identifiers indicate the same authority (the same host listening to the same TCP port). They are distinct name spaces and are considered to be distinct origin servers. However, an extension to HTTP that is defined to apply to entire host domains, such as the Cookie protocol [RFC6265], can allow information set by one service to impact communication with other services within a matching group of host domains.

The process for authoritative access to an "https" identified

resource is defined in [RFC2818].

2.7.3. http and https URI Normalization and Comparison

Since the "http" and "https" schemes conform to the URI generic syntax, such URIs are normalized and compared according to the algorithm defined in Section 6 of [RFC3986], using the defaults described above for each scheme.

If the port is equal to the default port for a scheme, the normal form is to omit the port subcomponent. When not being used in absolute form as the request target of an OPTIONS request, an empty path component is equivalent to an absolute path of "/", so the normal form is to provide a path of "/" instead. The scheme and host are case-insensitive and normally provided in lowercase; all other components are compared in a case-sensitive manner. Characters other than those in the "reserved" set are equivalent to their percent-encoded octets: the normal form is to not encode them (see Sections 2.1 and 2.2 of [RFC3986]).

For example, the following three URIs are equivalent:

```
http://example.com:80/~smith/home.html
http://EXAMPLE.com/%7Esmith/home.html
http://EXAMPLE.com:/%7esmith/home.html
```

3. Message Format

All HTTP/1.1 messages consist of a start-line followed by a sequence of octets in a format similar to the Internet Message Format [RFC5322]: zero or more header fields (collectively referred to as the "headers" or the "header section"), an empty line indicating the end of the header section, and an optional message body.

```
HTTP-message  = start-line
                *( header-field CRLF )
                CRLF
                [ message-body ]
```

The normal procedure for parsing an HTTP message is to read the start-line into a structure, read each header field into a hash table by field name until the empty line, and then use the parsed data to determine if a message body is expected. If a message body has been indicated, then it is read as a stream until an amount of octets equal to the message body length is read or the connection is closed.

A recipient MUST parse an HTTP message as a sequence of octets in an encoding that is a superset of US-ASCII [USASCII]. Parsing an HTTP

message as a stream of Unicode characters, without regard for the specific encoding, creates security vulnerabilities due to the varying ways that string processing libraries handle invalid multibyte character sequences that contain the octet LF (%x0A). String-based parsers can only be safely used within protocol elements after the element has been extracted from the message, such as within a header field-value after message parsing has delineated the individual fields.

An HTTP message can be parsed as a stream for incremental processing or forwarding downstream. However, recipients cannot rely on incremental delivery of partial messages, since some implementations will buffer or delay message forwarding for the sake of network efficiency, security checks, or payload transformations.

A sender **MUST NOT** send whitespace between the start-line and the first header field. A recipient that receives whitespace between the start-line and the first header field **MUST** either reject the message as invalid or consume each whitespace-preceded line without further processing of it (i.e., ignore the entire line, along with any subsequent lines preceded by whitespace, until a properly formed header field is received or the header section is terminated).

The presence of such whitespace in a request might be an attempt to trick a server into ignoring that field or processing the line after it as a new request, either of which might result in a security vulnerability if other implementations within the request chain interpret the same message differently. Likewise, the presence of such whitespace in a response might be ignored by some clients or cause others to cease parsing.

3.1. Start Line

An HTTP message can either be a request from client to server or a response from server to client. Syntactically, the two types of message differ only in the start-line, which is either a request-line (for requests) or a status-line (for responses), and in the algorithm for determining the length of the message body (Section 3.3).

In theory, a client could receive requests and a server could receive responses, distinguishing them by their different start-line formats, but in practice servers are implemented to only expect a request (a response is interpreted as an unknown or invalid request method) and clients are implemented to only expect a response.

start-line = request-line / status-line

3.1.1. Request Line

A request-line begins with a method token, followed by a single space (SP), the request-target, another single space (SP), the protocol version, and ending with CRLF.

```
request-line    = method SP request-target SP HTTP-version CRLF
```

The method token indicates the request method to be performed on the target resource. The request method is case-sensitive.

```
method          = token
```

The request methods defined by this specification can be found in Section 4 of [Part2], along with information regarding the HTTP method registry and considerations for defining new methods.

The request-target identifies the target resource upon which to apply the request, as defined in Section 5.3.

Recipients typically parse the request-line into its component parts by splitting on whitespace (see Section 3.5), since no whitespace is allowed in the three components. Unfortunately, some user agents fail to properly encode or exclude whitespace found in hypertext references, resulting in those disallowed characters being sent in a request-target.

Recipients of an invalid request-line SHOULD respond with either a 400 (Bad Request) error or a 301 (Moved Permanently) redirect with the request-target properly encoded. A recipient SHOULD NOT attempt to autocorrect and then process the request without a redirect, since the invalid request-line might be deliberately crafted to bypass security filters along the request chain.

HTTP does not place a pre-defined limit on the length of a request-line, as described in Section 2.5. A server that receives a method longer than any that it implements SHOULD respond with a 501 (Not Implemented) status code. A server that receives a request-target longer than any URI it wishes to parse MUST respond with a 414 (URI Too Long) status code (see Section 6.5.12 of [Part2]).

Various ad-hoc limitations on request-line length are found in practice. It is RECOMMENDED that all HTTP senders and recipients support, at a minimum, request-line lengths of 8000 octets.

3.1.2. Status Line

The first line of a response message is the status-line, consisting of the protocol version, a space (SP), the status code, another space, a possibly-empty textual phrase describing the status code, and ending with CRLF.

```
status-line = HTTP-version SP status-code SP reason-phrase CRLF
```

The status-code element is a 3-digit integer code describing the result of the server's attempt to understand and satisfy the client's corresponding request. The rest of the response message is to be interpreted in light of the semantics defined for that status code. See Section 6 of [Part2] for information about the semantics of status codes, including the classes of status code (indicated by the first digit), the status codes defined by this specification, considerations for the definition of new status codes, and the IANA registry.

```
status-code = 3DIGIT
```

The reason-phrase element exists for the sole purpose of providing a textual description associated with the numeric status code, mostly out of deference to earlier Internet application protocols that were more frequently used with interactive text clients. A client SHOULD ignore the reason-phrase content.

```
reason-phrase = *( HTAB / SP / VCHAR / obs-text )
```

3.2. Header Fields

Each header field consists of a case-insensitive field name followed by a colon (":"), optional leading whitespace, the field value, and optional trailing whitespace.

```
header-field = field-name ":" OWS field-value OWS
```

```
field-name = token
```

```
field-value = *( field-content / obs-fold )
```

```
field-content = field-vchar [ 1*( SP / HTAB ) field-vchar ]
```

```
field-vchar = VCHAR / obs-text
```

```
obs-fold = CRLF 1*( SP / HTAB )  
; obsolete line folding  
; see Section 3.2.4
```

The field-name token labels the corresponding field-value as having the semantics defined by that header field. For example, the Date

header field is defined in Section 7.1.1.2 of [Part2] as containing the origination timestamp for the message in which it appears.

3.2.1. Field Extensibility

Header fields are fully extensible: there is no limit on the introduction of new field names, each presumably defining new semantics, nor on the number of header fields used in a given message. Existing fields are defined in each part of this specification and in many other specifications outside this document set.

New header fields can be defined such that, when they are understood by a recipient, they might override or enhance the interpretation of previously defined header fields, define preconditions on request evaluation, or refine the meaning of responses.

A proxy **MUST** forward unrecognized header fields unless the field-name is listed in the Connection header field (Section 6.1) or the proxy is specifically configured to block, or otherwise transform, such fields. Other recipients **SHOULD** ignore unrecognized header fields. These requirements allow HTTP's functionality to be enhanced without requiring prior update of deployed intermediaries.

All defined header fields ought to be registered with IANA in the Message Header Field Registry, as described in Section 8.3 of [Part2].

3.2.2. Field Order

The order in which header fields with differing field names are received is not significant. However, it is good practice to send header fields that contain control data first, such as Host on requests and Date on responses, so that implementations can decide when not to handle a message as early as possible. A server **MUST NOT** apply a request to the target resource until the entire request header section is received, since later header fields might include conditionals, authentication credentials, or deliberately misleading duplicate header fields that would impact request processing.

A sender **MUST NOT** generate multiple header fields with the same field name in a message unless either the entire field value for that header field is defined as a comma-separated list [i.e., #(values)] or the header field is a well-known exception (as noted below).

A recipient **MAY** combine multiple header fields with the same field name into one "field-name: field-value" pair, without changing the semantics of the message, by appending each subsequent field value to

the combined field value in order, separated by a comma. The order in which header fields with the same field name are received is therefore significant to the interpretation of the combined field value; a proxy MUST NOT change the order of these field values when forwarding a message.

Note: In practice, the "Set-Cookie" header field ([RFC6265]) often appears multiple times in a response message and does not use the list syntax, violating the above requirements on multiple header fields with the same name. Since it cannot be combined into a single field-value, recipients ought to handle "Set-Cookie" as a special case while processing header fields. (See Appendix A.2.3 of [Kri2001] for details.)

3.2.3. Whitespace

This specification uses three rules to denote the use of linear whitespace: OWS (optional whitespace), RWS (required whitespace), and BWS ("bad" whitespace).

The OWS rule is used where zero or more linear whitespace octets might appear. For protocol elements where optional whitespace is preferred to improve readability, a sender SHOULD generate the optional whitespace as a single SP; otherwise, a sender SHOULD NOT generate optional whitespace except as needed to white-out invalid or unwanted protocol elements during in-place message filtering.

The RWS rule is used when at least one linear whitespace octet is required to separate field tokens. A sender SHOULD generate RWS as a single SP.

The BWS rule is used where the grammar allows optional whitespace only for historical reasons. A sender MUST NOT generate BWS in messages. A recipient MUST parse for such bad whitespace and remove it before interpreting the protocol element.

| | |
|-----|-----------------------|
| OWS | = *(SP / HTAB) |
| | ; optional whitespace |
| RWS | = 1*(SP / HTAB) |
| | ; required whitespace |
| BWS | = OWS |
| | ; "bad" whitespace |

3.2.4. Field Parsing

Messages are parsed using a generic algorithm, independent of the individual header field names. The contents within a given field value are not parsed until a later stage of message interpretation (usually after the message's entire header section has been processed). Consequently, this specification does not use ABNF rules to define each "Field-Name: Field Value" pair, as was done in previous editions. Instead, this specification uses ABNF rules which are named according to each registered field name, wherein the rule defines the valid grammar for that field's corresponding field values (i.e., after the field-value has been extracted from the header section by a generic field parser).

No whitespace is allowed between the header field-name and colon. In the past, differences in the handling of such whitespace have led to security vulnerabilities in request routing and response handling. A server **MUST** reject any received request message that contains whitespace between a header field-name and colon with a response code of 400 (Bad Request). A proxy **MUST** remove any such whitespace from a response message before forwarding the message downstream.

A field value might be preceded and/or followed by optional whitespace (OWS); a single SP preceding the field-value is preferred for consistent readability by humans. The field value does not include any leading or trailing white space: OWS occurring before the first non-whitespace octet of the field value or after the last non-whitespace octet of the field value ought to be excluded by parsers when extracting the field value from a header field.

Historically, HTTP header field values could be extended over multiple lines by preceding each extra line with at least one space or horizontal tab (obs-fold). This specification deprecates such line folding except within the message/http media type (Section 8.3.1). A sender **MUST NOT** generate a message that includes line folding (i.e., that has any field-value that contains a match to the obs-fold rule) unless the message is intended for packaging within the message/http media type.

A server that receives an obs-fold in a request message that is not within a message/http container **MUST** either reject the message by sending a 400 (Bad Request), preferably with a representation explaining that obsolete line folding is unacceptable, or replace each received obs-fold with one or more SP octets prior to interpreting the field value or forwarding the message downstream.

A proxy or gateway that receives an obs-fold in a response message that is not within a message/http container **MUST** either discard the

message and replace it with a 502 (Bad Gateway) response, preferably with a representation explaining that unacceptable line folding was received, or replace each received obs-fold with one or more SP octets prior to interpreting the field value or forwarding the message downstream.

A user agent that receives an obs-fold in a response message that is not within a message/http container MUST replace each received obs-fold with one or more SP octets prior to interpreting the field value.

Historically, HTTP has allowed field content with text in the ISO-8859-1 [ISO-8859-1] charset, supporting other charsets only through use of [RFC2047] encoding. In practice, most HTTP header field values use only a subset of the US-ASCII charset [USASCII]. Newly defined header fields SHOULD limit their field values to US-ASCII octets. A recipient SHOULD treat other octets in field content (obs-text) as opaque data.

3.2.5. Field Limits

HTTP does not place a pre-defined limit on the length of each header field or on the length of the header section as a whole, as described in Section 2.5. Various ad-hoc limitations on individual header field length are found in practice, often depending on the specific field semantics.

A server that receives a request header field, or set of fields, larger than it wishes to process MUST respond with an appropriate 4xx (Client Error) status code. Ignoring such header fields would increase the server's vulnerability to request smuggling attacks (Section 9.5).

A client MAY discard or truncate received header fields that are larger than the client wishes to process if the field semantics are such that the dropped value(s) can be safely ignored without changing the message framing or response semantics.

3.2.6. Field value components

Most HTTP header field values are defined using common syntax components (token, quoted-string, and comment) separated by whitespace or specific delimiting characters. Delimiters are chosen from the set of US-ASCII visual characters not allowed in a token (DQUOTE and "(),/:;<=>?@[\\]{}").

```
token          = 1*tchar

tchar          = "!" / "#" / "$" / "%" / "&" / "'" / "*"
                / "+" / "-" / "." / "^" / "_" / "`" / "|" / "~"
                / DIGIT / ALPHA
                ; any VCHAR, except delimiters
```

A string of text is parsed as a single value if it is quoted using double-quote marks.

```
quoted-string  = DQUOTE *( qdtext / quoted-pair ) DQUOTE
qdtext         = HTAB / SP / %x21 / %x23-5B / %x5D-7E / obs-text
obs-text       = %x80-FF
```

Comments can be included in some HTTP header fields by surrounding the comment text with parentheses. Comments are only allowed in fields containing "comment" as part of their field value definition.

```
comment        = "(" *( ctext / quoted-pair / comment ) ")"
ctext          = HTAB / SP / %x21-27 / %x2A-5B / %x5D-7E / obs-text
```

The backslash octet ("\") can be used as a single-octet quoting mechanism within quoted-string and comment constructs. Recipients that process the value of a quoted-string MUST handle a quoted-pair as if it were replaced by the octet following the backslash.

```
quoted-pair    = "\" ( HTAB / SP / VCHAR / obs-text )
```

A sender SHOULD NOT generate a quoted-pair in a quoted-string except where necessary to quote DQUOTE and backslash octets occurring within that string. A sender SHOULD NOT generate a quoted-pair in a comment except where necessary to quote parentheses ["(" and ")"] and backslash octets occurring within that comment.

3.3. Message Body

The message body (if any) of an HTTP message is used to carry the payload body of that request or response. The message body is identical to the payload body unless a transfer coding has been applied, as described in Section 3.3.1.

```
message-body = *OCTET
```

The rules for when a message body is allowed in a message differ for requests and responses.

The presence of a message body in a request is signaled by a Content-Length or Transfer-Encoding header field. Request message framing is

independent of method semantics, even if the method does not define any use for a message body.

The presence of a message body in a response depends on both the request method to which it is responding and the response status code (Section 3.1.2). Responses to the HEAD request method (Section 4.3.2 of [Part2]) never include a message body because the associated response header fields (e.g., Transfer-Encoding, Content-Length, etc.), if present, indicate only what their values would have been if the request method had been GET (Section 4.3.1 of [Part2]). 2xx (Successful) responses to a CONNECT request method (Section 4.3.6 of [Part2]) switch to tunnel mode instead of having a message body. All 1xx (Informational), 204 (No Content), and 304 (Not Modified) responses do not include a message body. All other responses do include a message body, although the body might be of zero length.

3.3.1. Transfer-Encoding

The Transfer-Encoding header field lists the transfer coding names corresponding to the sequence of transfer codings that have been (or will be) applied to the payload body in order to form the message body. Transfer codings are defined in Section 4.

Transfer-Encoding = 1#transfer-coding

Transfer-Encoding is analogous to the Content-Transfer-Encoding field of MIME, which was designed to enable safe transport of binary data over a 7-bit transport service ([RFC2045], Section 6). However, safe transport has a different focus for an 8bit-clean transfer protocol. In HTTP's case, Transfer-Encoding is primarily intended to accurately delimit a dynamically generated payload and to distinguish payload encodings that are only applied for transport efficiency or security from those that are characteristics of the selected resource.

A recipient MUST be able to parse the chunked transfer coding (Section 4.1) because it plays a crucial role in framing messages when the payload body size is not known in advance. A sender MUST NOT apply chunked more than once to a message body (i.e., chunking an already chunked message is not allowed). If any transfer coding other than chunked is applied to a request payload body, the sender MUST apply chunked as the final transfer coding to ensure that the message is properly framed. If any transfer coding other than chunked is applied to a response payload body, the sender MUST either apply chunked as the final transfer coding or terminate the message by closing the connection.

For example,

Transfer-Encoding: gzip, chunked

indicates that the payload body has been compressed using the gzip coding and then chunked using the chunked coding while forming the message body.

Unlike Content-Encoding (Section 3.1.2.1 of [Part2]), Transfer-Encoding is a property of the message, not of the representation, and any recipient along the request/response chain MAY decode the received transfer coding(s) or apply additional transfer coding(s) to the message body, assuming that corresponding changes are made to the Transfer-Encoding field-value. Additional information about the encoding parameters can be provided by other header fields not defined by this specification.

Transfer-Encoding MAY be sent in a response to a HEAD request or in a 304 (Not Modified) response (Section 4.1 of [Part4]) to a GET request, neither of which includes a messagebody, to indicate that the origin server would have applied a transfer coding to the message body if the request had been an unconditional GET. This indication is not required, however, because any recipient on the response chain (including the origin server) can remove transfer codings when they are not needed.

A server MUST NOT send a Transfer-Encoding header field in any response with a status code of 1xx (Informational) or 204 (No Content). A server MUST NOT send a Transfer-Encoding header field in any 2xx (Successful) response to a CONNECT request (Section 4.3.6 of [Part2]).

Transfer-Encoding was added in HTTP/1.1. It is generally assumed that implementations advertising only HTTP/1.0 support will not understand how to process a transfer-encoded payload. A client MUST NOT send a request containing Transfer-Encoding unless it knows the server will handle HTTP/1.1 (or later) requests; such knowledge might be in the form of specific user configuration or by remembering the version of a prior received response. A server MUST NOT send a response containing Transfer-Encoding unless the corresponding request indicates HTTP/1.1 (or later).

A server that receives a request message with a transfer coding it does not understand SHOULD respond with 501 (Not Implemented).

3.3.2. Content-Length

When a message does not have a Transfer-Encoding header field, a Content-Length header field can provide the anticipated size, as a decimal number of octets, for a potential payload body. For messages that do include a payload body, the Content-Length field-value provides the framing information necessary for determining where the body (and message) ends. For messages that do not include a payload body, the Content-Length indicates the size of the selected representation (Section 3 of [Part2]).

Content-Length = 1*DIGIT

An example is

Content-Length: 3495

A sender **MUST NOT** send a Content-Length header field in any message that contains a Transfer-Encoding header field.

A user agent **SHOULD** send a Content-Length in a request message when no Transfer-Encoding is sent and the request method defines a meaning for an enclosed payload body. For example, a Content-Length header field is normally sent in a POST request even when the value is 0 (indicating an empty payload body). A user agent **SHOULD NOT** send a Content-Length header field when the request message does not contain a payload body and the method semantics do not anticipate such a body.

A server **MAY** send a Content-Length header field in a response to a HEAD request (Section 4.3.2 of [Part2]); a server **MUST NOT** send Content-Length in such a response unless its field-value equals the decimal number of octets that would have been sent in the payload body of a response if the same request had used the GET method.

A server **MAY** send a Content-Length header field in a 304 (Not Modified) response to a conditional GET request (Section 4.1 of [Part4]); a server **MUST NOT** send Content-Length in such a response unless its field-value equals the decimal number of octets that would have been sent in the payload body of a 200 (OK) response to the same request.

A server **MUST NOT** send a Content-Length header field in any response with a status code of 1xx (Informational) or 204 (No Content). A server **MUST NOT** send a Content-Length header field in any 2xx (Successful) response to a CONNECT request (Section 4.3.6 of [Part2]).

Aside from the cases defined above, in the absence of Transfer-Encoding, an origin server SHOULD send a Content-Length header field when the payload body size is known prior to sending the complete header section. This will allow downstream recipients to measure transfer progress, know when a received message is complete, and potentially reuse the connection for additional requests.

Any Content-Length field value greater than or equal to zero is valid. Since there is no predefined limit to the length of a payload, a recipient MUST anticipate potentially large decimal numerals and prevent parsing errors due to integer conversion overflows (Section 9.3).

If a message is received that has multiple Content-Length header fields with field-values consisting of the same decimal value, or a single Content-Length header field with a field value containing a list of identical decimal values (e.g., "Content-Length: 42, 42"), indicating that duplicate Content-Length header fields have been generated or combined by an upstream message processor, then the recipient MUST either reject the message as invalid or replace the duplicated field-values with a single valid Content-Length field containing that decimal value prior to determining the message body length or forwarding the message.

Note: HTTP's use of Content-Length for message framing differs significantly from the same field's use in MIME, where it is an optional field used only within the "message/external-body" media-type.

3.3.3. Message Body Length

The length of a message body is determined by one of the following (in order of precedence):

1. Any response to a HEAD request and any response with a 1xx (Informational), 204 (No Content), or 304 (Not Modified) status code is always terminated by the first empty line after the header fields, regardless of the header fields present in the message, and thus cannot contain a message body.
2. Any 2xx (Successful) response to a CONNECT request implies that the connection will become a tunnel immediately after the empty line that concludes the header fields. A client MUST ignore any Content-Length or Transfer-Encoding header fields received in such a message.
3. If a Transfer-Encoding header field is present and the chunked transfer coding (Section 4.1) is the final encoding, the message

body length is determined by reading and decoding the chunked data until the transfer coding indicates the data is complete.

If a Transfer-Encoding header field is present in a response and the chunked transfer coding is not the final encoding, the message body length is determined by reading the connection until it is closed by the server. If a Transfer-Encoding header field is present in a request and the chunked transfer coding is not the final encoding, the message body length cannot be determined reliably; the server MUST respond with the 400 (Bad Request) status code and then close the connection.

If a message is received with both a Transfer-Encoding and a Content-Length header field, the Transfer-Encoding overrides the Content-Length. Such a message might indicate an attempt to perform request smuggling (Section 9.5) or response splitting (Section 9.4) and ought to be handled as an error. A sender MUST remove the received Content-Length field prior to forwarding such a message downstream.

4. If a message is received without Transfer-Encoding and with either multiple Content-Length header fields having differing field-values or a single Content-Length header field having an invalid value, then the message framing is invalid and the recipient MUST treat it as an unrecoverable error. If this is a request message, the server MUST respond with a 400 (Bad Request) status code and then close the connection. If this is a response message received by a proxy, the proxy MUST close the connection to the server, discard the received response, and send a 502 (Bad Gateway) response to the client. If this is a response message received by a user agent, the user agent MUST close the connection to the server and discard the received response.
5. If a valid Content-Length header field is present without Transfer-Encoding, its decimal value defines the expected message body length in octets. If the sender closes the connection or the recipient times out before the indicated number of octets are received, the recipient MUST consider the message to be incomplete and close the connection.
6. If this is a request message and none of the above are true, then the message body length is zero (no message body is present).
7. Otherwise, this is a response message without a declared message body length, so the message body length is determined by the number of octets received prior to the server closing the connection.

Since there is no way to distinguish a successfully completed, close-delimited message from a partially-received message interrupted by network failure, a server SHOULD generate encoding or length-delimited messages whenever possible. The close-delimiting feature exists primarily for backwards compatibility with HTTP/1.0.

A server MAY reject a request that contains a message body but not a Content-Length by responding with 411 (Length Required).

Unless a transfer coding other than chunked has been applied, a client that sends a request containing a message body SHOULD use a valid Content-Length header field if the message body length is known in advance, rather than the chunked transfer coding, since some existing services respond to chunked with a 411 (Length Required) status code even though they understand the chunked transfer coding. This is typically because such services are implemented via a gateway that requires a content-length in advance of being called and the server is unable or unwilling to buffer the entire request before processing.

A user agent that sends a request containing a message body MUST send a valid Content-Length header field if it does not know the server will handle HTTP/1.1 (or later) requests; such knowledge can be in the form of specific user configuration or by remembering the version of a prior received response.

If the final response to the last request on a connection has been completely received and there remains additional data to read, a user agent MAY discard the remaining data or attempt to determine if that data belongs as part of the prior response body, which might be the case if the prior message's Content-Length value is incorrect. A client MUST NOT process, cache, or forward such extra data as a separate response, since such behavior would be vulnerable to cache poisoning.

3.4. Handling Incomplete Messages

A server that receives an incomplete request message, usually due to a canceled request or a triggered time-out exception, MAY send an error response prior to closing the connection.

A client that receives an incomplete response message, which can occur when a connection is closed prematurely or when decoding a supposedly chunked transfer coding fails, MUST record the message as incomplete. Cache requirements for incomplete responses are defined in Section 3 of [Part6].

If a response terminates in the middle of the header section (before

the empty line is received) and the status code might rely on header fields to convey the full meaning of the response, then the client cannot assume that meaning has been conveyed; the client might need to repeat the request in order to determine what action to take next.

A message body that uses the chunked transfer coding is incomplete if the zero-sized chunk that terminates the encoding has not been received. A message that uses a valid Content-Length is incomplete if the size of the message body received (in octets) is less than the value given by Content-Length. A response that has neither chunked transfer coding nor Content-Length is terminated by closure of the connection, and thus is considered complete regardless of the number of message body octets received, provided that the header section was received intact.

3.5. Message Parsing Robustness

Older HTTP/1.0 user agent implementations might send an extra CRLF after a POST request as a workaround for some early server applications that failed to read message body content that was not terminated by a line-ending. An HTTP/1.1 user agent **MUST NOT** preface or follow a request with an extra CRLF. If terminating the request message body with a line-ending is desired, then the user agent **MUST** count the terminating CRLF octets as part of the message body length.

In the interest of robustness, a server that is expecting to receive and parse a request-line **SHOULD** ignore at least one empty line (CRLF) received prior to the request-line.

Although the line terminator for the start-line and header fields is the sequence CRLF, a recipient **MAY** recognize a single LF as a line terminator and ignore any preceding CR.

Although the request-line and status-line grammar rules require that each of the component elements be separated by a single SP octet, recipients **MAY** instead parse on whitespace-delimited word boundaries and, aside from the CRLF terminator, treat any form of whitespace as the SP separator while ignoring preceding or trailing whitespace; such whitespace includes one or more of the following octets: SP, HTAB, VT (%x0B), FF (%x0C), or bare CR. However, lenient parsing can result in security vulnerabilities if there are multiple recipients of the message and each has its own unique interpretation of robustness (see Section 9.5).

When a server listening only for HTTP request messages, or processing what appears from the start-line to be an HTTP request message, receives a sequence of octets that does not match the HTTP-message grammar aside from the robustness exceptions listed above, the server

SHOULD respond with a 400 (Bad Request) response.

4. Transfer Codings

Transfer coding names are used to indicate an encoding transformation that has been, can be, or might need to be applied to a payload body in order to ensure "safe transport" through the network. This differs from a content coding in that the transfer coding is a property of the message rather than a property of the representation that is being transferred.

```
transfer-coding    = "chunked" ; Section 4.1
                  / "compress" ; Section 4.2.1
                  / "deflate" ; Section 4.2.2
                  / "gzip" ; Section 4.2.3
                  / transfer-extension
transfer-extension = token *( OWS ";" OWS transfer-parameter )
```

Parameters are in the form of a name or name=value pair.

```
transfer-parameter = token BWS "=" BWS ( token / quoted-string )
```

All transfer-coding names are case-insensitive and ought to be registered within the HTTP Transfer Coding registry, as defined in Section 8.4. They are used in the TE (Section 4.3) and Transfer-Encoding (Section 3.3.1) header fields.

4.1. Chunked Transfer Coding

The chunked transfer coding wraps the payload body in order to transfer it as a series of chunks, each with its own size indicator, followed by an OPTIONAL trailer containing header fields. Chunked enables content streams of unknown size to be transferred as a sequence of length-delimited buffers, which enables the sender to retain connection persistence and the recipient to know when it has received the entire message.

```
chunked-body      = *chunk
                  last-chunk
                  trailer-part
                  CRLF

chunk              = chunk-size [ chunk-ext ] CRLF
                  chunk-data CRLF
chunk-size         = 1*HEXDIG
last-chunk         = 1*("0") [ chunk-ext ] CRLF

chunk-data         = 1*OCTET ; a sequence of chunk-size octets
```

The chunk-size field is a string of hex digits indicating the size of the chunk-data in octets. The chunked transfer coding is complete when a chunk with a chunk-size of zero is received, possibly followed by a trailer, and finally terminated by an empty line.

A recipient **MUST** be able to parse and decode the chunked transfer coding.

4.1.1. Chunk Extensions

The chunked encoding allows each chunk to include zero or more chunk extensions, immediately following the chunk-size, for the sake of supplying per-chunk metadata (such as a signature or hash), mid-message control information, or randomization of message body size.

```
chunk-ext      = *( ";" chunk-ext-name [ "=" chunk-ext-val ] )
```

```
chunk-ext-name = token
```

```
chunk-ext-val  = token / quoted-string
```

The chunked encoding is specific to each connection and is likely to be removed or recoded by each recipient (including intermediaries) before any higher-level application would have a chance to inspect the extensions. Hence, use of chunk extensions is generally limited to specialized HTTP services such as "long polling" (where client and server can have shared expectations regarding the use of chunk extensions) or for padding within an end-to-end secured connection.

A recipient **MUST** ignore unrecognized chunk extensions. A server ought to limit the total length of chunk extensions received in a request to an amount reasonable for the services provided, in the same way that it applies length limitations and timeouts for other parts of a message, and generate an appropriate 4xx (Client Error) response if that amount is exceeded.

4.1.2. Chunked Trailer Part

A trailer allows the sender to include additional fields at the end of a chunked message in order to supply metadata that might be dynamically generated while the message body is sent, such as a message integrity check, digital signature, or post-processing status. The trailer fields are identical to header fields, except they are sent in a chunked trailer instead of the message's header section.

```
trailer-part   = *( header-field CRLF )
```

A sender **MUST NOT** generate a trailer that contains a field necessary

for message framing (e.g., Transfer-Encoding and Content-Length), routing (e.g., Host), request modifiers (e.g., controls and conditionals in Section 5 of [Part2]), authentication (e.g., see [Part7] and [RFC6265]), response control data (e.g., see Section 7.1 of [Part2]), or determining how to process the payload (e.g., Content-Encoding, Content-Type, Content-Range, and Trailer).

When a chunked message containing a non-empty trailer is received, the recipient MAY process the fields (aside from those forbidden above) as if they were appended to the message's header section. A recipient MUST ignore (or consider as an error) any fields that are forbidden to be sent in a trailer, since processing them as if they were present in the header section might bypass external security filters.

Unless the request includes a TE header field indicating "trailers" is acceptable, as described in Section 4.3, a server SHOULD NOT generate trailer fields that it believes are necessary for the user agent to receive. Without a TE containing "trailers", the server ought to assume that the trailer fields might be silently discarded along the path to the user agent. This requirement allows intermediaries to forward a de-chunked message to an HTTP/1.0 recipient without buffering the entire response.

4.1.3. Decoding Chunked

A process for decoding the chunked transfer coding can be represented in pseudo-code as:

```
length := 0
read chunk-size, chunk-ext (if any), and CRLF
while (chunk-size > 0) {
    read chunk-data and CRLF
    append chunk-data to decoded-body
    length := length + chunk-size
    read chunk-size, chunk-ext (if any), and CRLF
}
read trailer field
while (trailer field is not empty) {
    if (trailer field is allowed to be sent in a trailer) {
        append trailer field to existing header fields
    }
    read trailer-field
}
Content-Length := length
Remove "chunked" from Transfer-Encoding
Remove Trailer from existing header fields
```

4.2. Compression Codings

The codings defined below can be used to compress the payload of a message.

4.2.1. Compress Coding

The "compress" coding is an adaptive Lempel-Ziv-Welch (LZW) coding [Welch] that is commonly produced by the UNIX file compression program "compress". A recipient SHOULD consider "x-compress" to be equivalent to "compress".

4.2.2. Deflate Coding

The "deflate" coding is a "zlib" data format [RFC1950] containing a "deflate" compressed data stream [RFC1951] that uses a combination of the Lempel-Ziv (LZ77) compression algorithm and Huffman coding.

Note: Some non-conformant implementations send the "deflate" compressed data without the zlib wrapper.

4.2.3. Gzip Coding

The "gzip" coding is an LZ77 coding with a 32 bit CRC that is commonly produced by the gzip file compression program [RFC1952]. A recipient SHOULD consider "x-gzip" to be equivalent to "gzip".

4.3. TE

The "TE" header field in a request indicates what transfer codings, besides chunked, the client is willing to accept in response, and whether or not the client is willing to accept trailer fields in a chunked transfer coding.

The TE field-value consists of a comma-separated list of transfer coding names, each allowing for optional parameters (as described in Section 4), and/or the keyword "trailers". A client MUST NOT send the chunked transfer coding name in TE; chunked is always acceptable for HTTP/1.1 recipients.

```
TE           = #t-codings
t-codings    = "trailers" / ( transfer-coding [ t-ranking ] )
t-ranking    = OWS ";" OWS "q=" rank
rank         = ( "0" [ "." 0*3DIGIT ] )
              / ( "1" [ "." 0*3("0") ] )
```

Three examples of TE use are below.

```
TE: deflate
TE:
TE: trailers, deflate;q=0.5
```

The presence of the keyword "trailers" indicates that the client is willing to accept trailer fields in a chunked transfer coding, as defined in Section 4.1.2, on behalf of itself and any downstream clients. For requests from an intermediary, this implies that either: (a) all downstream clients are willing to accept trailer fields in the forwarded response; or, (b) the intermediary will attempt to buffer the response on behalf of downstream recipients. Note that HTTP/1.1 does not define any means to limit the size of a chunked response such that an intermediary can be assured of buffering the entire response.

When multiple transfer codings are acceptable, the client MAY rank the codings by preference using a case-insensitive "q" parameter (similar to the qvalues used in content negotiation fields, Section 5.3.1 of [Part2]). The rank value is a real number in the range 0 through 1, where 0.001 is the least preferred and 1 is the most preferred; a value of 0 means "not acceptable".

If the TE field-value is empty or if no TE field is present, the only acceptable transfer coding is chunked. A message with no transfer coding is always acceptable.

Since the TE header field only applies to the immediate connection, a sender of TE MUST also send a "TE" connection option within the Connection header field (Section 6.1) in order to prevent the TE field from being forwarded by intermediaries that do not support its semantics.

4.4. Trailer

When a message includes a message body encoded with the chunked transfer coding and the sender desires to send metadata in the form of trailer fields at the end of the message, the sender SHOULD generate a Trailer header field before the message body to indicate which fields will be present in the trailers. This allows the recipient to prepare for receipt of that metadata before it starts processing the body, which is useful if the message is being streamed and the recipient wishes to confirm an integrity check on the fly.

```
Trailer = 1#field-name
```

5. Message Routing

HTTP request message routing is determined by each client based on the target resource, the client's proxy configuration, and establishment or reuse of an inbound connection. The corresponding response routing follows the same connection chain back to the client.

5.1. Identifying a Target Resource

HTTP is used in a wide variety of applications, ranging from general-purpose computers to home appliances. In some cases, communication options are hard-coded in a client's configuration. However, most HTTP clients rely on the same resource identification mechanism and configuration techniques as general-purpose Web browsers.

HTTP communication is initiated by a user agent for some purpose. The purpose is a combination of request semantics, which are defined in [Part2], and a target resource upon which to apply those semantics. A URI reference (Section 2.7) is typically used as an identifier for the "target resource", which a user agent would resolve to its absolute form in order to obtain the "target URI". The target URI excludes the reference's fragment component, if any, since fragment identifiers are reserved for client-side processing ([RFC3986], Section 3.5).

5.2. Connecting Inbound

Once the target URI is determined, a client needs to decide whether a network request is necessary to accomplish the desired semantics and, if so, where that request is to be directed.

If the client has a cache [Part6] and the request can be satisfied by it, then the request is usually directed there first.

If the request is not satisfied by a cache, then a typical client will check its configuration to determine whether a proxy is to be used to satisfy the request. Proxy configuration is implementation-dependent, but is often based on URI prefix matching, selective authority matching, or both, and the proxy itself is usually identified by an "http" or "https" URI. If a proxy is applicable, the client connects inbound by establishing (or reusing) a connection to that proxy.

If no proxy is applicable, a typical client will invoke a handler routine, usually specific to the target URI's scheme, to connect directly to an authority for the target resource. How that is accomplished is dependent on the target URI scheme and defined by its

associated specification, similar to how this specification defines origin server access for resolution of the "http" (Section 2.7.1) and "https" (Section 2.7.2) schemes.

HTTP requirements regarding connection management are defined in Section 6.

5.3. Request Target

Once an inbound connection is obtained, the client sends an HTTP request message (Section 3) with a request-target derived from the target URI. There are four distinct formats for the request-target, depending on both the method being requested and whether the request is to a proxy.

```
request-target = origin-form
                / absolute-form
                / authority-form
                / asterisk-form
```

5.3.1. origin-form

The most common form of request-target is the origin-form.

```
origin-form    = absolute-path [ "?" query ]
```

When making a request directly to an origin server, other than a CONNECT or server-wide OPTIONS request (as detailed below), a client MUST send only the absolute path and query components of the target URI as the request-target. If the target URI's path component is empty, the client MUST send "/" as the path within the origin-form of request-target. A Host header field is also sent, as defined in Section 5.4.

For example, a client wishing to retrieve a representation of the resource identified as

```
http://www.example.org/where?q=now
```

directly from the origin server would open (or reuse) a TCP connection to port 80 of the host "www.example.org" and send the lines:

```
GET /where?q=now HTTP/1.1
Host: www.example.org
```

followed by the remainder of the request message.

5.3.2. absolute-form

When making a request to a proxy, other than a CONNECT or server-wide OPTIONS request (as detailed below), a client MUST send the target URI in absolute-form as the request-target.

absolute-form = absolute-URI

The proxy is requested to either service that request from a valid cache, if possible, or make the same request on the client's behalf to either the next inbound proxy server or directly to the origin server indicated by the request-target. Requirements on such "forwarding" of messages are defined in Section 5.7.

An example absolute-form of request-line would be:

```
GET http://www.example.org/pub/WWW/TheProject.html HTTP/1.1
```

To allow for transition to the absolute-form for all requests in some future version of HTTP, a server MUST accept the absolute-form in requests, even though HTTP/1.1 clients will only send them in requests to proxies.

5.3.3. authority-form

The authority-form of request-target is only used for CONNECT requests (Section 4.3.6 of [Part2]).

authority-form = authority

When making a CONNECT request to establish a tunnel through one or more proxies, a client MUST send only the target URI's authority component (excluding any userinfo and its "@" delimiter) as the request-target. For example,

```
CONNECT www.example.com:80 HTTP/1.1
```

5.3.4. asterisk-form

The asterisk-form of request-target is only used for a server-wide OPTIONS request (Section 4.3.7 of [Part2]).

asterisk-form = "*"

When a client wishes to request OPTIONS for the server as a whole, as opposed to a specific named resource of that server, the client MUST send only "*" (%x2A) as the request-target. For example,

OPTIONS * HTTP/1.1

If a proxy receives an OPTIONS request with an absolute-form of request-target in which the URI has an empty path and no query component, then the last proxy on the request chain MUST send a request-target of "*" when it forwards the request to the indicated origin server.

For example, the request

```
OPTIONS http://www.example.org:8001 HTTP/1.1
```

would be forwarded by the final proxy as

```
OPTIONS * HTTP/1.1
Host: www.example.org:8001
```

after connecting to port 8001 of host "www.example.org".

5.4. Host

The "Host" header field in a request provides the host and port information from the target URI, enabling the origin server to distinguish among resources while servicing requests for multiple host names on a single IP address.

Host = uri-host [":" port] ; Section 2.7.1

A client MUST send a Host header field in all HTTP/1.1 request messages. If the target URI includes an authority component, then a client MUST send a field-value for Host that is identical to that authority component, excluding any userinfo subcomponent and its "@" delimiter (Section 2.7.1). If the authority component is missing or undefined for the target URI, then a client MUST send a Host header field with an empty field-value.

Since the Host field-value is critical information for handling a request, a user agent SHOULD generate Host as the first header field following the request-line.

For example, a GET request to the origin server for <http://www.example.org/pub/WWW/> would begin with:

```
GET /pub/WWW/ HTTP/1.1
Host: www.example.org
```

A client MUST send a Host header field in an HTTP/1.1 request even if the request-target is in the absolute-form, since this allows the

Host information to be forwarded through ancient HTTP/1.0 proxies that might not have implemented Host.

When a proxy receives a request with an absolute-form of request-target, the proxy MUST ignore the received Host header field (if any) and instead replace it with the host information of the request-target. A proxy that forwards such a request MUST generate a new Host field-value based on the received request-target rather than forward the received Host field-value.

Since the Host header field acts as an application-level routing mechanism, it is a frequent target for malware seeking to poison a shared cache or redirect a request to an unintended server. An interception proxy is particularly vulnerable if it relies on the Host field-value for redirecting requests to internal servers, or for use as a cache key in a shared cache, without first verifying that the intercepted connection is targeting a valid IP address for that host.

A server MUST respond with a 400 (Bad Request) status code to any HTTP/1.1 request message that lacks a Host header field and to any request message that contains more than one Host header field or a Host header field with an invalid field-value.

5.5. Effective Request URI

Since the request-target often contains only part of the user agent's target URI, a server reconstructs the intended target as an "effective request URI" to properly service the request. This reconstruction involves both the server's local configuration and information communicated in the request-target, Host header field, and connection context.

For a user agent, the effective request URI is the target URI.

If the request-target is in absolute-form, the effective request URI is the same as the request-target. Otherwise, the effective request URI is constructed as follows:

If the server's configuration (or outbound gateway) provides a fixed URI scheme, that scheme is used for the effective request URI. Otherwise, if the request is received over a TLS-secured TCP connection, the effective request URI's scheme is "https"; if not, the scheme is "http".

If the server's configuration (or outbound gateway) provides a fixed URI authority component, that authority is used for the effective request URI. If not, then if the request-target is in

authority-form, the effective request URI's authority component is the same as the request-target. If not, then if a Host header field is supplied with a non-empty field-value, the authority component is the same as the Host field-value. Otherwise, the authority component is assigned the default name configured for the server and, if the connection's incoming TCP port number differs from the default port for the effective request URI's scheme, then a colon (":") and the incoming port number (in decimal form) are appended to the authority component.

If the request-target is in authority-form or asterisk-form, the effective request URI's combined path and query component is empty. Otherwise, the combined path and query component is the same as the request-target.

The components of the effective request URI, once determined as above, can be combined into absolute-URI form by concatenating the scheme, "://", authority, and combined path and query component.

Example 1: the following message received over an insecure TCP connection

```
GET /pub/WWW/TheProject.html HTTP/1.1
Host: www.example.org:8080
```

has an effective request URI of

```
http://www.example.org:8080/pub/WWW/TheProject.html
```

Example 2: the following message received over a TLS-secured TCP connection

```
OPTIONS * HTTP/1.1
Host: www.example.org
```

has an effective request URI of

```
https://www.example.org
```

Recipients of an HTTP/1.0 request that lacks a Host header field might need to use heuristics (e.g., examination of the URI path for something unique to a particular host) in order to guess the effective request URI's authority component.

Once the effective request URI has been constructed, an origin server needs to decide whether or not to provide service for that URI via the connection in which the request was received. For example, the request might have been misdirected, deliberately or accidentally,

such that the information within a received request-target or Host header field differs from the host or port upon which the connection has been made. If the connection is from a trusted gateway, that inconsistency might be expected; otherwise, it might indicate an attempt to bypass security filters, trick the server into delivering non-public content, or poison a cache. See Section 9 for security considerations regarding message routing.

5.6. Associating a Response to a Request

HTTP does not include a request identifier for associating a given request message with its corresponding one or more response messages. Hence, it relies on the order of response arrival to correspond exactly to the order in which requests are made on the same connection. More than one response message per request only occurs when one or more informational responses (1xx, see Section 6.2 of [Part2]) precede a final response to the same request.

A client that has more than one outstanding request on a connection MUST maintain a list of outstanding requests in the order sent and MUST associate each received response message on that connection to the highest ordered request that has not yet received a final (non-1xx) response.

5.7. Message Forwarding

As described in Section 2.3, intermediaries can serve a variety of roles in the processing of HTTP requests and responses. Some intermediaries are used to improve performance or availability. Others are used for access control or to filter content. Since an HTTP stream has characteristics similar to a pipe-and-filter architecture, there are no inherent limits to the extent an intermediary can enhance (or interfere) with either direction of the stream.

An intermediary not acting as a tunnel MUST implement the Connection header field, as specified in Section 6.1, and exclude fields from being forwarded that are only intended for the incoming connection.

An intermediary MUST NOT forward a message to itself unless it is protected from an infinite request loop. In general, an intermediary ought to recognize its own server names, including any aliases, local variations, or literal IP addresses, and respond to such requests directly.

5.7.1. Via

The "Via" header field indicates the presence of intermediate protocols and recipients between the user agent and the server (on requests) or between the origin server and the client (on responses), similar to the "Received" header field in email (Section 3.6.7 of [RFC5322]). Via can be used for tracking message forwards, avoiding request loops, and identifying the protocol capabilities of senders along the request/response chain.

```
Via = 1#( received-protocol RWS received-by [ RWS comment ] )
```

```
received-protocol = [ protocol-name "/" ] protocol-version
                  ; see Section 6.7
received-by       = ( uri-host [ ":" port ] ) / pseudonym
pseudonym         = token
```

Multiple Via field values represent each proxy or gateway that has forwarded the message. Each intermediary appends its own information about how the message was received, such that the end result is ordered according to the sequence of forwarding recipients.

A proxy **MUST** send an appropriate Via header field, as described below, in each message that it forwards. An HTTP-to-HTTP gateway **MUST** send an appropriate Via header field in each inbound request message and **MAY** send a Via header field in forwarded response messages.

For each intermediary, the received-protocol indicates the protocol and protocol version used by the upstream sender of the message. Hence, the Via field value records the advertised protocol capabilities of the request/response chain such that they remain visible to downstream recipients; this can be useful for determining what backwards-incompatible features might be safe to use in response, or within a later request, as described in Section 2.6. For brevity, the protocol-name is omitted when the received protocol is HTTP.

The received-by portion of the field value is normally the host and optional port number of a recipient server or client that subsequently forwarded the message. However, if the real host is considered to be sensitive information, a sender **MAY** replace it with a pseudonym. If a port is not provided, a recipient **MAY** interpret that as meaning it was received on the default TCP port, if any, for the received-protocol.

A sender **MAY** generate comments in the Via header field to identify the software of each recipient, analogous to the User-Agent and

Server header fields. However, all comments in the Via field are optional and a recipient MAY remove them prior to forwarding the message.

For example, a request message could be sent from an HTTP/1.0 user agent to an internal proxy code-named "fred", which uses HTTP/1.1 to forward the request to a public proxy at p.example.net, which completes the request by forwarding it to the origin server at www.example.com. The request received by www.example.com would then have the following Via header field:

```
Via: 1.0 fred, 1.1 p.example.net
```

An intermediary used as a portal through a network firewall SHOULD NOT forward the names and ports of hosts within the firewall region unless it is explicitly enabled to do so. If not enabled, such an intermediary SHOULD replace each received-by host of any host behind the firewall by an appropriate pseudonym for that host.

An intermediary MAY combine an ordered subsequence of Via header field entries into a single such entry if the entries have identical received-protocol values. For example,

```
Via: 1.0 ricky, 1.1 ethel, 1.1 fred, 1.0 lucy
```

could be collapsed to

```
Via: 1.0 ricky, 1.1 mertz, 1.0 lucy
```

A sender SHOULD NOT combine multiple entries unless they are all under the same organizational control and the hosts have already been replaced by pseudonyms. A sender MUST NOT combine entries that have different received-protocol values.

5.7.2. Transformations

Some intermediaries include features for transforming messages and their payloads. A proxy might, for example, convert between image formats in order to save cache space or to reduce the amount of traffic on a slow link. However, operational problems might occur when these transformations are applied to payloads intended for critical applications, such as medical imaging or scientific data analysis, particularly when integrity checks or digital signatures are used to ensure that the payload received is identical to the original.

An HTTP-to-HTTP proxy is called a "transforming proxy" if it is designed or configured to modify messages in a semantically

meaningful way (i.e., modifications, beyond those required by normal HTTP processing, that change the message in a way that would be significant to the original sender or potentially significant to downstream recipients). For example, a transforming proxy might be acting as a shared annotation server (modifying responses to include references to a local annotation database), a malware filter, a format transcoder, or a privacy filter. Such transformations are presumed to be desired by whichever client (or client organization) selected the proxy.

If a proxy receives a request-target with a host name that is not a fully qualified domain name, it MAY add its own domain to the host name it received when forwarding the request. A proxy MUST NOT change the host name if the request-target contains a fully qualified domain name.

A proxy MUST NOT modify the "absolute-path" and "query" parts of the received request-target when forwarding it to the next inbound server, except as noted above to replace an empty path with "/" or "*".

A proxy MAY modify the message body through application or removal of a transfer coding (Section 4).

A proxy MUST NOT transform the payload (Section 3.3 of [Part2]) of a message that contains a no-transform cache-control directive (Section 5.2 of [Part6]).

A proxy MAY transform the payload of a message that does not contain a no-transform cache-control directive. A proxy that transforms a payload MUST add a Warning header field with the warn-code of 214 ("Transformation Applied") if one is not already in the message (see Section 5.5 of [Part6]). A proxy that transforms the payload of a 200 (OK) response can further inform downstream recipients that a transformation has been applied by changing the response status code to 203 (Non-Authoritative Information) (Section 6.3.4 of [Part2]).

A proxy SHOULD NOT modify header fields that provide information about the end points of the communication chain, the resource state, or the selected representation (other than the payload) unless the field's definition specifically allows such modification or the modification is deemed necessary for privacy or security.

6. Connection Management

HTTP messaging is independent of the underlying transport or session-layer connection protocol(s). HTTP only presumes a reliable transport with in-order delivery of requests and the corresponding

in-order delivery of responses. The mapping of HTTP request and response structures onto the data units of an underlying transport protocol is outside the scope of this specification.

As described in Section 5.2, the specific connection protocols to be used for an HTTP interaction are determined by client configuration and the target URI. For example, the "http" URI scheme (Section 2.7.1) indicates a default connection of TCP over IP, with a default TCP port of 80, but the client might be configured to use a proxy via some other connection, port, or protocol.

HTTP implementations are expected to engage in connection management, which includes maintaining the state of current connections, establishing a new connection or reusing an existing connection, processing messages received on a connection, detecting connection failures, and closing each connection. Most clients maintain multiple connections in parallel, including more than one connection per server endpoint. Most servers are designed to maintain thousands of concurrent connections, while controlling request queues to enable fair use and detect denial of service attacks.

6.1. Connection

The "Connection" header field allows the sender to indicate desired control options for the current connection. In order to avoid confusing downstream recipients, a proxy or gateway **MUST** remove or replace any received connection options before forwarding the message.

When a header field aside from Connection is used to supply control information for or about the current connection, the sender **MUST** list the corresponding field-name within the "Connection" header field. A proxy or gateway **MUST** parse a received Connection header field before a message is forwarded and, for each connection-option in this field, remove any header field(s) from the message with the same name as the connection-option, and then remove the Connection header field itself (or replace it with the intermediary's own connection options for the forwarded message).

Hence, the Connection header field provides a declarative way of distinguishing header fields that are only intended for the immediate recipient ("hop-by-hop") from those fields that are intended for all recipients on the chain ("end-to-end"), enabling the message to be self-descriptive and allowing future connection-specific extensions to be deployed without fear that they will be blindly forwarded by older intermediaries.

The Connection header field's value has the following grammar:

Connection = 1#connection-option
connection-option = token

Connection options are case-insensitive.

A sender **MUST NOT** send a connection option corresponding to a header field that is intended for all recipients of the payload. For example, Cache-Control is never appropriate as a connection option (Section 5.2 of [Part6]).

The connection options do not always correspond to a header field present in the message, since a connection-specific header field might not be needed if there are no parameters associated with a connection option. In contrast, a connection-specific header field that is received without a corresponding connection option usually indicates that the field has been improperly forwarded by an intermediary and ought to be ignored by the recipient.

When defining new connection options, specification authors ought to survey existing header field names and ensure that the new connection option does not share the same name as an already deployed header field. Defining a new connection option essentially reserves that potential field-name for carrying additional information related to the connection option, since it would be unwise for senders to use that field-name for anything else.

The "close" connection option is defined for a sender to signal that this connection will be closed after completion of the response. For example,

Connection: close

in either the request or the response header fields indicates that the sender is going to close the connection after the current request/response is complete (Section 6.6).

A client that does not support persistent connections **MUST** send the "close" connection option in every request message.

A server that does not support persistent connections **MUST** send the "close" connection option in every response message that does not have a 1xx (Informational) status code.

6.2. Establishment

It is beyond the scope of this specification to describe how connections are established via various transport or session-layer protocols. Each connection applies to only one transport link.

6.3. Persistence

HTTP/1.1 defaults to the use of "persistent connections", allowing multiple requests and responses to be carried over a single connection. The "close" connection-option is used to signal that a connection will not persist after the current request/response. HTTP implementations SHOULD support persistent connections.

A recipient determines whether a connection is persistent or not based on the most recently received message's protocol version and Connection header field (if any):

- o If the close connection option is present, the connection will not persist after the current response; else,
- o If the received protocol is HTTP/1.1 (or later), the connection will persist after the current response; else,
- o If the received protocol is HTTP/1.0, the "keep-alive" connection option is present, the recipient is not a proxy, and the recipient wishes to honor the HTTP/1.0 "keep-alive" mechanism, the connection will persist after the current response; otherwise,
- o The connection will close after the current response.

A client MAY send additional requests on a persistent connection until it sends or receives a close connection option or receives an HTTP/1.0 response without a "keep-alive" connection option.

In order to remain persistent, all messages on a connection need to have a self-defined message length (i.e., one not defined by closure of the connection), as described in Section 3.3. A server MUST read the entire request message body or close the connection after sending its response, since otherwise the remaining data on a persistent connection would be misinterpreted as the next request. Likewise, a client MUST read the entire response message body if it intends to reuse the same connection for a subsequent request.

A proxy server MUST NOT maintain a persistent connection with an HTTP/1.0 client (see Section 19.7.1 of [RFC2068] for information and discussion of the problems with the Keep-Alive header field implemented by many HTTP/1.0 clients).

See Appendix A.1.2 for more information on backward compatibility with HTTP/1.0 clients.

6.3.1. Retrying Requests

Connections can be closed at any time, with or without intention. Implementations ought to anticipate the need to recover from asynchronous close events.

When an inbound connection is closed prematurely, a client MAY open a new connection and automatically retransmit an aborted sequence of requests if all of those requests have idempotent methods (Section 4.2.2 of [Part2]). A proxy MUST NOT automatically retry non-idempotent requests.

A user agent MUST NOT automatically retry a request with a non-idempotent method unless it has some means to know that the request semantics are actually idempotent, regardless of the method, or some means to detect that the original request was never applied. For example, a user agent that knows (through design or configuration) that a POST request to a given resource is safe can repeat that request automatically. Likewise, a user agent designed specifically to operate on a version control repository might be able to recover from partial failure conditions by checking the target resource revision(s) after a failed connection, reverting or fixing any changes that were partially applied, and then automatically retrying the requests that failed.

A client SHOULD NOT automatically retry a failed automatic retry.

6.3.2. Pipelining

A client that supports persistent connections MAY "pipeline" its requests (i.e., send multiple requests without waiting for each response). A server MAY process a sequence of pipelined requests in parallel if they all have safe methods (Section 4.2.1 of [Part2]), but MUST send the corresponding responses in the same order that the requests were received.

A client that pipelines requests SHOULD retry unanswered requests if the connection closes before it receives all of the corresponding responses. When retrying pipelined requests after a failed connection (a connection not explicitly closed by the server in its last complete response), a client MUST NOT pipeline immediately after connection establishment, since the first remaining request in the prior pipeline might have caused an error response that can be lost again if multiple requests are sent on a prematurely closed connection (see the TCP reset problem described in Section 6.6).

Idempotent methods (Section 4.2.2 of [Part2]) are significant to pipelining because they can be automatically retried after a

connection failure. A user agent SHOULD NOT pipeline requests after a non-idempotent method, until the final response status code for that method has been received, unless the user agent has a means to detect and recover from partial failure conditions involving the pipelined sequence.

An intermediary that receives pipelined requests MAY pipeline those requests when forwarding them inbound, since it can rely on the outbound user agent(s) to determine what requests can be safely pipelined. If the inbound connection fails before receiving a response, the pipelining intermediary MAY attempt to retry a sequence of requests that have yet to receive a response if the requests all have idempotent methods; otherwise, the pipelining intermediary SHOULD forward any received responses and then close the corresponding outbound connection(s) so that the outbound user agent(s) can recover accordingly.

6.4. Concurrency

A client ought to limit the number of simultaneous open connections that it maintains to a given server.

Previous revisions of HTTP gave a specific number of connections as a ceiling, but this was found to be impractical for many applications. As a result, this specification does not mandate a particular maximum number of connections, but instead encourages clients to be conservative when opening multiple connections.

Multiple connections are typically used to avoid the "head-of-line blocking" problem, wherein a request that takes significant server-side processing and/or has a large payload blocks subsequent requests on the same connection. However, each connection consumes server resources. Furthermore, using multiple connections can cause undesirable side effects in congested networks.

Note that a server might reject traffic that it deems abusive or characteristic of a denial of service attack, such as an excessive number of open connections from a single client.

6.5. Failures and Time-outs

Servers will usually have some time-out value beyond which they will no longer maintain an inactive connection. Proxy servers might make this a higher value since it is likely that the client will be making more connections through the same proxy server. The use of persistent connections places no requirements on the length (or existence) of this time-out for either the client or the server.

A client or server that wishes to time-out **SHOULD** issue a graceful close on the connection. Implementations **SHOULD** constantly monitor open connections for a received closure signal and respond to it as appropriate, since prompt closure of both sides of a connection enables allocated system resources to be reclaimed.

A client, server, or proxy **MAY** close the transport connection at any time. For example, a client might have started to send a new request at the same time that the server has decided to close the "idle" connection. From the server's point of view, the connection is being closed while it was idle, but from the client's point of view, a request is in progress.

A server **SHOULD** sustain persistent connections, when possible, and allow the underlying transport's flow control mechanisms to resolve temporary overloads, rather than terminate connections with the expectation that clients will retry. The latter technique can exacerbate network congestion.

A client sending a message body **SHOULD** monitor the network connection for an error response while it is transmitting the request. If the client sees a response that indicates the server does not wish to receive the message body and is closing the connection, the client **SHOULD** immediately cease transmitting the body and close its side of the connection.

6.6. Tear-down

The Connection header field (Section 6.1) provides a "close" connection option that a sender **SHOULD** send when it wishes to close the connection after the current request/response pair.

A client that sends a close connection option **MUST NOT** send further requests on that connection (after the one containing close) and **MUST** close the connection after reading the final response message corresponding to this request.

A server that receives a close connection option **MUST** initiate a close of the connection (see below) after it sends the final response to the request that contained close. The server **SHOULD** send a close connection option in its final response on that connection. The server **MUST NOT** process any further requests received on that connection.

A server that sends a close connection option **MUST** initiate a close of the connection (see below) after it sends the response containing close. The server **MUST NOT** process any further requests received on that connection.

A client that receives a close connection option MUST cease sending requests on that connection and close the connection after reading the response message containing the close; if additional pipelined requests had been sent on the connection, the client SHOULD NOT assume that they will be processed by the server.

If a server performs an immediate close of a TCP connection, there is a significant risk that the client will not be able to read the last HTTP response. If the server receives additional data from the client on a fully-closed connection, such as another request that was sent by the client before receiving the server's response, the server's TCP stack will send a reset packet to the client; unfortunately, the reset packet might erase the client's unacknowledged input buffers before they can be read and interpreted by the client's HTTP parser.

To avoid the TCP reset problem, servers typically close a connection in stages. First, the server performs a half-close by closing only the write side of the read/write connection. The server then continues to read from the connection until it receives a corresponding close by the client, or until the server is reasonably certain that its own TCP stack has received the client's acknowledgement of the packet(s) containing the server's last response. Finally, the server fully closes the connection.

It is unknown whether the reset problem is exclusive to TCP or might also be found in other transport connection protocols.

6.7. Upgrade

The "Upgrade" header field is intended to provide a simple mechanism for transitioning from HTTP/1.1 to some other protocol on the same connection. A client MAY send a list of protocols in the Upgrade header field of a request to invite the server to switch to one or more of those protocols, in order of descending preference, before sending the final response. A server MAY ignore a received Upgrade header field if it wishes to continue using the current protocol on that connection. Upgrade cannot be used to insist on a protocol change.

Upgrade = 1#protocol

protocol = protocol-name ["/" protocol-version]
protocol-name = token
protocol-version = token

A server that sends a 101 (Switching Protocols) response MUST send an Upgrade header field to indicate the new protocol(s) to which the

connection is being switched; if multiple protocol layers are being switched, the sender MUST list the protocols in layer-ascending order. A server MUST NOT switch to a protocol that was not indicated by the client in the corresponding request's Upgrade header field. A server MAY choose to ignore the order of preference indicated by the client and select the new protocol(s) based on other factors, such as the nature of the request or the current load on the server.

A server that sends a 426 (Upgrade Required) response MUST send an Upgrade header field to indicate the acceptable protocols, in order of descending preference.

A server MAY send an Upgrade header field in any other response to advertise that it implements support for upgrading to the listed protocols, in order of descending preference, when appropriate for a future request.

The following is a hypothetical example sent by a client:

```
GET /hello.txt HTTP/1.1
Host: www.example.com
Connection: upgrade
Upgrade: HTTP/2.0, SHTTP/1.3, IRC/6.9, RTA/x11
```

The capabilities and nature of the application-level communication after the protocol change is entirely dependent upon the new protocol(s) chosen. However, immediately after sending the 101 response, the server is expected to continue responding to the original request as if it had received its equivalent within the new protocol (i.e., the server still has an outstanding request to satisfy after the protocol has been changed, and is expected to do so without requiring the request to be repeated).

For example, if the Upgrade header field is received in a GET request and the server decides to switch protocols, it first responds with a 101 (Switching Protocols) message in HTTP/1.1 and then immediately follows that with the new protocol's equivalent of a response to a GET on the target resource. This allows a connection to be upgraded to protocols with the same semantics as HTTP without the latency cost of an additional round-trip. A server MUST NOT switch protocols unless the received message semantics can be honored by the new protocol; an OPTIONS request can be honored by any protocol.

The following is an example response to the above hypothetical request:

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

[... data stream switches to HTTP/2.0 with an appropriate response (as defined by new protocol) to the "GET /hello.txt" request ...]

When Upgrade is sent, the sender MUST also send a Connection header field (Section 6.1) that contains an "upgrade" connection option, in order to prevent Upgrade from being accidentally forwarded by intermediaries that might not implement the listed protocols. A server MUST ignore an Upgrade header field that is received in an HTTP/1.0 request.

A client cannot begin using an upgraded protocol on the connection until it has completely sent the request message (i.e., the client can't change the protocol it is sending in the middle of a message). If a server receives both Upgrade and an Expect header field with the "100-continue" expectation (Section 5.1.1 of [Part2]), the server MUST send a 100 (Continue) response before sending a 101 (Switching Protocols) response.

The Upgrade header field only applies to switching protocols on top of the existing connection; it cannot be used to switch the underlying connection (transport) protocol, nor to switch the existing communication to a different connection. For those purposes, it is more appropriate to use a 3xx (Redirection) response (Section 6.4 of [Part2]).

This specification only defines the protocol name "HTTP" for use by the family of Hypertext Transfer Protocols, as defined by the HTTP version rules of Section 2.6 and future updates to this specification. Additional tokens ought to be registered with IANA using the registration procedure defined in Section 8.6.

7. ABNF list extension: #rule

A #rule extension to the ABNF rules of [RFC5234] is used to improve readability in the definitions of some header field values.

A construct "#" is defined, similar to "*", for defining comma-delimited lists of elements. The full form is "<n>#<m>element" indicating at least <n> and at most <m> elements, each separated by a single comma (",") and optional whitespace (OWS).

In any production that uses the list construct, a sender MUST NOT generate empty list elements. In other words, a sender MUST generate lists that satisfy the following syntax:

```
1#element => element *( OWS "," OWS element )
```

and:

```
#element => [ 1#element ]
```

and for $n \geq 1$ and $m > 1$:

```
<n>#<m>element => element <n-1>*<m-1>( OWS "," OWS element )
```

For compatibility with legacy list rules, a recipient MUST parse and ignore a reasonable number of empty list elements: enough to handle common mistakes by senders that merge values, but not so much that they could be used as a denial of service mechanism. In other words, a recipient MUST accept lists that satisfy the following syntax:

```
#element => [ ( "," / element ) *( OWS "," [ OWS element ] ) ]
```

```
1#element => *( "," OWS ) element *( OWS "," [ OWS element ] )
```

Empty elements do not contribute to the count of elements present. For example, given these ABNF productions:

```
example-list      = 1#example-list-elmt
example-list-elmt = token ; see Section 3.2.6
```

Then the following are valid values for example-list (not including the double quotes, which are present for delimitation only):

```
"foo,bar"
"foo ,bar,"
"foo , ,bar,charlie  "
```

In contrast, the following values would be invalid, since at least one non-empty element is required by the example-list production:

```
" "
" ,"
" , , , "
```

Appendix B shows the collected ABNF for recipients after the list constructs have been expanded.

8. IANA Considerations

8.1. Header Field Registration

HTTP header fields are registered within the Message Header Field Registry maintained at
<<http://www.iana.org/assignments/message-headers/>>.

This document defines the following HTTP header fields, so their associated registry entries shall be updated according to the permanent registrations below (see [BCP90]):

| Header Field Name | Protocol | Status | Reference |
|-------------------|----------|----------|---------------|
| Connection | http | standard | Section 6.1 |
| Content-Length | http | standard | Section 3.3.2 |
| Host | http | standard | Section 5.4 |
| TE | http | standard | Section 4.3 |
| Trailer | http | standard | Section 4.4 |
| Transfer-Encoding | http | standard | Section 3.3.1 |
| Upgrade | http | standard | Section 6.7 |
| Via | http | standard | Section 5.7.1 |

Furthermore, the header field-name "Close" shall be registered as "reserved", since using that name as an HTTP header field might conflict with the "close" connection option of the "Connection" header field (Section 6.1).

| Header Field Name | Protocol | Status | Reference |
|-------------------|----------|----------|-------------|
| Close | http | reserved | Section 8.1 |

The change controller is: "IETF (iesg@ietf.org) - Internet Engineering Task Force".

8.2. URI Scheme Registration

IANA maintains the registry of URI Schemes [BCP115] at
<<http://www.iana.org/assignments/uri-schemes/>>.

This document defines the following URI schemes, so their associated registry entries shall be updated according to the permanent registrations below:

| URI Scheme | Description | Reference |
|------------|------------------------------------|---------------|
| http | Hypertext Transfer Protocol | Section 2.7.1 |
| https | Hypertext Transfer Protocol Secure | Section 2.7.2 |

8.3. Internet Media Type Registration

IANA maintains the registry of Internet media types [BCP13] at <http://www.iana.org/assignments/media-types>.

This document serves as the specification for the Internet media types "message/http" and "application/http". The following is to be registered with IANA.

8.3.1. Internet Media Type message/http

The message/http type can be used to enclose a single HTTP request or response message, provided that it obeys the MIME restrictions for all "message" types regarding line length and encodings.

Type name: message

Subtype name: http

Required parameters: N/A

Optional parameters: version, msgtype

version: The HTTP-version number of the enclosed message (e.g., "1.1"). If not present, the version can be determined from the first line of the body.

msgtype: The message type -- "request" or "response". If not present, the type can be determined from the first line of the body.

Encoding considerations: only "7bit", "8bit", or "binary" are permitted

Security considerations: see Section 9

Interoperability considerations: N/A

Published specification: This specification (see Section 8.3.1).

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information:

Magic number(s): N/A

Deprecated alias names for this type: N/A

File extension(s): N/A

Macintosh file type code(s): N/A

Person and email address to contact for further information: See
Authors Section.

Intended usage: COMMON

Restrictions on usage: N/A

Author: See Authors Section.

Change controller: IESG

8.3.2. Internet Media Type application/http

The application/http type can be used to enclose a pipeline of one or more HTTP request or response messages (not intermixed).

Type name: application

Subtype name: http

Required parameters: N/A

Optional parameters: version, msgtype

version: The HTTP-version number of the enclosed messages (e.g., "1.1"). If not present, the version can be determined from the first line of the body.

msgtype: The message type -- "request" or "response". If not present, the type can be determined from the first line of the body.

Encoding considerations: HTTP messages enclosed by this type are in "binary" format; use of an appropriate Content-Transfer-Encoding is required when transmitted via E-mail.

Security considerations: see Section 9

Interoperability considerations: N/A

Published specification: This specification (see Section 8.3.2).

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information:

Deprecated alias names for this type: N/A

Magic number(s): N/A

File extension(s): N/A

Macintosh file type code(s): N/A

Person and email address to contact for further information: See Authors Section.

Intended usage: COMMON

Restrictions on usage: N/A

Author: See Authors Section.

Change controller: IESG

8.4. Transfer Coding Registry

The HTTP Transfer Coding Registry defines the name space for transfer coding names. It is maintained at <http://www.iana.org/assignments/http-parameters>.

8.4.1. Procedure

Registrations MUST include the following fields:

- o Name

- o Description
- o Pointer to specification text

Names of transfer codings MUST NOT overlap with names of content codings (Section 3.1.2.1 of [Part2]) unless the encoding transformation is identical, as is the case for the compression codings defined in Section 4.2.

Values to be added to this name space require IETF Review (see Section 4.1 of [RFC5226]), and MUST conform to the purpose of transfer coding defined in this specification.

Use of program names for the identification of encoding formats is not desirable and is discouraged for future encodings.

8.4.2. Registration

The HTTP Transfer Coding Registry shall be updated with the registrations below:

| Name | Description | Reference |
|------------|---|---------------|
| chunked | Transfer in a series of chunks | Section 4.1 |
| compress | UNIX "compress" data format [Welch] | Section 4.2.1 |
| deflate | "deflate" compressed data ([RFC1951]) inside the "zlib" data format ([RFC1950]) | Section 4.2.2 |
| gzip | GZIP file format [RFC1952] | Section 4.2.3 |
| x-compress | Deprecated (alias for compress) | Section 4.2.1 |
| x-gzip | Deprecated (alias for gzip) | Section 4.2.3 |

8.5. Content Coding Registration

IANA maintains the registry of HTTP Content Codings at <http://www.iana.org/assignments/http-parameters>.

The HTTP Content Codings Registry shall be updated with the registrations below:

| Name | Description | Reference |
|------------|---|---------------|
| compress | UNIX "compress" data format [Welch] | Section 4.2.1 |
| deflate | "deflate" compressed data ([RFC1951]) inside the "zlib" data format ([RFC1950]) | Section 4.2.2 |
| gzip | GZIP file format [RFC1952] | Section 4.2.3 |
| x-compress | Deprecated (alias for compress) | Section 4.2.1 |
| x-gzip | Deprecated (alias for gzip) | Section 4.2.3 |

8.6. Upgrade Token Registry

The HTTP Upgrade Token Registry defines the name space for protocol-name tokens used to identify protocols in the Upgrade header field. The registry is maintained at <http://www.iana.org/assignments/http-upgrade-tokens>.

8.6.1. Procedure

Each registered protocol name is associated with contact information and an optional set of specifications that details how the connection will be processed after it has been upgraded.

Registrations happen on a "First Come First Served" basis (see Section 4.1 of [RFC5226]) and are subject to the following rules:

1. A protocol-name token, once registered, stays registered forever.
2. The registration MUST name a responsible party for the registration.
3. The registration MUST name a point of contact.
4. The registration MAY name a set of specifications associated with that token. Such specifications need not be publicly available.
5. The registration SHOULD name a set of expected "protocol-version" tokens associated with that token at the time of registration.
6. The responsible party MAY change the registration at any time. The IANA will keep a record of all such changes, and make them available upon request.
7. The IESG MAY reassign responsibility for a protocol token. This will normally only be used in the case when a responsible party cannot be contacted.

This registration procedure for HTTP Upgrade Tokens replaces that previously defined in Section 7.2 of [RFC2817].

8.6.2. Upgrade Token Registration

The "HTTP" entry in the HTTP Upgrade Token Registry shall be updated with the registration below:

| Value | Description | Expected Version Tokens | Reference |
|-------|-----------------------------|------------------------------|-------------|
| HTTP | Hypertext Transfer Protocol | any DIGIT.DIGIT (e.g, "2.0") | Section 2.6 |

The responsible party is: "IETF (iesg@ietf.org) - Internet Engineering Task Force".

9. Security Considerations

This section is meant to inform developers, information providers, and users of known security considerations relevant to HTTP message syntax, parsing, and routing. Security considerations about HTTP semantics and payloads are addressed in [Part2].

9.1. Establishing Authority

HTTP relies on the notion of an authoritative response: a response that has been determined by (or at the direction of) the authority identified within the target URI to be the most appropriate response for that request given the state of the target resource at the time of response message origination. Providing a response from a non-authoritative source, such as a shared cache, is often useful to improve performance and availability, but only to the extent that the source can be trusted or the distrusted response can be safely used.

Unfortunately, establishing authority can be difficult. For example, phishing is an attack on the user's perception of authority, where that perception can be misled by presenting similar branding in hypertext, possibly aided by userinfo obfuscating the authority component (see Section 2.7.1). User agents can reduce the impact of phishing attacks by enabling users to easily inspect a target URI prior to making an action, by prominently distinguishing (or rejecting) userinfo when present, and by not sending stored credentials and cookies when the referring document is from an unknown or untrusted source.

When a registered name is used in the authority component, the "http" URI scheme (Section 2.7.1) relies on the user's local name resolution service to determine where it can find authoritative responses. This means that any attack on a user's network host table, cached names, or name resolution libraries becomes an avenue for attack on establishing authority. Likewise, the user's choice of server for Domain Name Service (DNS), and the hierarchy of servers from which it obtains resolution results, could impact the authenticity of address mappings; DNSSEC ([RFC4033]) is one way to improve authenticity.

Furthermore, after an IP address is obtained, establishing authority for an "http" URI is vulnerable to attacks on Internet Protocol routing.

The "https" scheme (Section 2.7.2) is intended to prevent (or at least reveal) many of these potential attacks on establishing authority, provided that the negotiated TLS connection is secured and the client properly verifies that the communicating server's identity matches the target URI's authority component (see [RFC2818]). Correctly implementing such verification can be difficult (see [Georgiev]).

9.2. Risks of Intermediaries

By their very nature, HTTP intermediaries are men-in-the-middle, and thus represent an opportunity for man-in-the-middle attacks. Compromise of the systems on which the intermediaries run can result in serious security and privacy problems. Intermediaries might have access to security-related information, personal information about individual users and organizations, and proprietary information belonging to users and content providers. A compromised intermediary, or an intermediary implemented or configured without regard to security and privacy considerations, might be used in the commission of a wide range of potential attacks.

Intermediaries that contain a shared cache are especially vulnerable to cache poisoning attacks, as described in Section 8 of [Part6].

Implementers need to consider the privacy and security implications of their design and coding decisions, and of the configuration options they provide to operators (especially the default configuration).

Users need to be aware that intermediaries are no more trustworthy than the people who run them; HTTP itself cannot solve this problem.

9.3. Attacks via Protocol Element Length

Because HTTP uses mostly textual, character-delimited fields, parsers are often vulnerable to attacks based on sending very long (or very slow) streams of data, particularly where an implementation is expecting a protocol element with no predefined length.

To promote interoperability, specific recommendations are made for minimum size limits on request-line (Section 3.1.1) and header fields (Section 3.2). These are minimum recommendations, chosen to be supportable even by implementations with limited resources; it is expected that most implementations will choose substantially higher limits.

A server can reject a message that has a request-target that is too long (Section 6.5.12 of [Part2]) or a request payload that is too large (Section 6.5.11 of [Part2]). Additional status codes related to capacity limits have been defined by extensions to HTTP [RFC6585].

Recipients ought to carefully limit the extent to which they process other protocol elements, including (but not limited to) request methods, response status phrases, header field-names, numeric values, and body chunks. Failure to limit such processing can result in buffer overflows, arithmetic overflows, or increased vulnerability to denial of service attacks.

9.4. Response Splitting

Response splitting (a.k.a, CRLF injection) is a common technique, used in various attacks on Web usage, that exploits the line-based nature of HTTP message framing and the ordered association of requests to responses on persistent connections [Klein]. This technique can be particularly damaging when the requests pass through a shared cache.

Response splitting exploits a vulnerability in servers (usually within an application server) where an attacker can send encoded data within some parameter of the request that is later decoded and echoed within any of the response header fields of the response. If the decoded data is crafted to look like the response has ended and a subsequent response has begun, the response has been split and the content within the apparent second response is controlled by the attacker. The attacker can then make any other request on the same persistent connection and trick the recipients (including intermediaries) into believing that the second half of the split is an authoritative answer to the second request.

For example, a parameter within the request-target might be read by

an application server and reused within a redirect, resulting in the same parameter being echoed in the Location header field of the response. If the parameter is decoded by the application and not properly encoded when placed in the response field, the attacker can send encoded CRLF octets and other content that will make the application's single response look like two or more responses.

A common defense against response splitting is to filter requests for data that looks like encoded CR and LF (e.g., "%0D" and "%0A"). However, that assumes the application server is only performing URI decoding, rather than more obscure data transformations like charset transcoding, XML entity translation, base64 decoding, sprintf reformatting, etc. A more effective mitigation is to prevent anything other than the server's core protocol libraries from sending a CR or LF within the header section, which means restricting the output of header fields to APIs that filter for bad octets and not allowing application servers to write directly to the protocol stream.

9.5. Request Smuggling

Request smuggling ([Linhart]) is a technique that exploits differences in protocol parsing among various recipients to hide additional requests (which might otherwise be blocked or disabled by policy) within an apparently harmless request. Like response splitting, request smuggling can lead to a variety of attacks on HTTP usage.

This specification has introduced new requirements on request parsing, particularly with regard to message framing in Section 3.3.3, to reduce the effectiveness of request smuggling.

9.6. Message Integrity

HTTP does not define a specific mechanism for ensuring message integrity, instead relying on the error-detection ability of underlying transport protocols and the use of length or chunk-delimited framing to detect completeness. Additional integrity mechanisms, such as hash functions or digital signatures applied to the content, can be selectively added to messages via extensible metadata header fields. Historically, the lack of a single integrity mechanism has been justified by the informal nature of most HTTP communication. However, the prevalence of HTTP as an information access mechanism has resulted in its increasing use within environments where verification of message integrity is crucial.

User agents are encouraged to implement configurable means for detecting and reporting failures of message integrity such that those

means can be enabled within environments for which integrity is necessary. For example, a browser being used to view medical history or drug interaction information needs to indicate to the user when such information is detected by the protocol to be incomplete, expired, or corrupted during transfer. Such mechanisms might be selectively enabled via user agent extensions or the presence of message integrity metadata in a response. At a minimum, user agents ought to provide some indication that allows a user to distinguish between a complete and incomplete response message (Section 3.4) when such verification is desired.

9.7. Message Confidentiality

HTTP relies on underlying transport protocols to provide message confidentiality when that is desired. HTTP has been specifically designed to be independent of the transport protocol, such that it can be used over many different forms of encrypted connection, with the selection of such transports being identified by the choice of URI scheme or within user agent configuration.

The "https" scheme can be used to identify resources that require a confidential connection, as described in Section 2.7.2.

9.8. Privacy of Server Log Information

A server is in the position to save personal data about a user's requests over time, which might identify their reading patterns or subjects of interest. In particular, log information gathered at an intermediary often contains a history of user agent interaction, across a multitude of sites, that can be traced to individual users.

HTTP log information is confidential in nature; its handling is often constrained by laws and regulations. Log information needs to be securely stored and appropriate guidelines followed for its analysis. Anonymization of personal information within individual entries helps, but is generally not sufficient to prevent real log traces from being re-identified based on correlation with other access characteristics. As such, access traces that are keyed to a specific client are unsafe to publish even if the key is pseudonymous.

To minimize the risk of theft or accidental publication, log information ought to be purged of personally identifiable information, including user identifiers, IP addresses, and user-provided query parameters, as soon as that information is no longer necessary to support operational needs for security, auditing, or fraud control.

10. Acknowledgments

This edition of HTTP/1.1 builds on the many contributions that went into RFC 1945, RFC 2068, RFC 2145, and RFC 2616, including substantial contributions made by the previous authors, editors, and working group chairs: Tim Berners-Lee, Ari Luotonen, Roy T. Fielding, Henrik Frystyk Nielsen, Jim Gettys, Jeffrey C. Mogul, Larry Masinter, and Paul J. Leach. Mark Nottingham oversaw this effort as working group chair.

Since 1999, the following contributors have helped improve the HTTP specification by reporting bugs, asking smart questions, drafting or reviewing text, and evaluating open issues:

Adam Barth, Adam Roach, Addison Phillips, Adrian Chadd, Adrian Cole, Adrien W. de Croy, Alan Ford, Alan Ruttenberg, Albert Lunde, Alek Storm, Alex Rousskov, Alexandre Morgaut, Alexey Melnikov, Alisha Smith, Amichai Rothman, Amit Klein, Amos Jeffries, Andreas Maier, Andreas Petersson, Andrei Popov, Anil Sharma, Anne van Kesteren, Anthony Bryan, Asbjorn Ulsberg, Ashok Kumar, Balachander Krishnamurthy, Barry Leiba, Ben Laurie, Benjamin Carlyle, Benjamin Niven-Jenkins, Benoit Claise, Bil Corry, Bill Burke, Bjoern Hoehrmann, Bob Scheifler, Boris Zbarsky, Brett Slatkin, Brian Kell, Brian McBarron, Brian Pane, Brian Raymor, Brian Smith, Bruce Perens, Bryce Nesbitt, Cameron Heavon-Jones, Carl Kugler, Carsten Bormann, Charles Fry, Chris Burdess, Chris Newman, Christian Huitema, Cyrus Daboo, Dale Robert Anderson, Dan Wing, Dan Winship, Daniel Stenberg, Darrel Miller, Dave Cridland, Dave Crocker, Dave Kristol, Dave Thaler, David Booth, David Singer, David W. Morris, Diwakar Shetty, Dmitry Kurochkin, Drummond Reed, Duane Wessels, Edward Lee, Eitan Adler, Eliot Lear, Emile Stephan, Eran Hammer-Lahav, Eric D. Williams, Eric J. Bowman, Eric Lawrence, Eric Rescorla, Erik Aronesty, EungJun Yi, Evan Prodromou, Felix Geisendoerfer, Florian Weimer, Frank Ellermann, Fred Akalin, Fred Bohle, Frederic Kayser, Gabor Molnar, Gabriel Montenegro, Geoffrey Sneddon, Gervase Markham, Gili Tzabari, Grahame Grieve, Greg Slepak, Greg Wilkins, Grzegorz Calkowski, Harald Tveit Alvestrand, Harry Halpin, Helge Hess, Henrik Nordstrom, Henry S. Thompson, Henry Story, Herbert van de Sompel, Herve Ruellan, Howard Melman, Hugo Haas, Ian Fette, Ian Hickson, Ido Safriti, Ilari Liusvaara, Ilya Grigorik, Ingo Struck, J. Ross Nicoll, James Cloos, James H. Manger, James Lacey, James M. Snell, Jamie Lokier, Jan Algermissen, Jari Arkko, Jeff Hodges (who came up with the term 'effective Request-URI'), Jeff Pinner, Jeff Walden, Jim Luther, Jitu Padhye, Joe D. Williams, Joe Gregorio, Joe Orton, Joel Jaeggli, John C. Klensin, John C. Mallery, John Cowan, John Kemp, John Panzer, John Schneider, John Stracke, John Sullivan, Jonas Sicking, Jonathan A. Rees, Jonathan Billington, Jonathan Moore, Jonathan Silvera, Jordi Ros, Joris Dobbeltstein, Josh Cohen, Julien

Pierre, Jungshik Shin, Justin Chapweske, Justin Erenkrantz, Justin James, Kalvinder Singh, Karl Dubost, Kathleen Moriarty, Keith Hoffman, Keith Moore, Ken Murchison, Koen Holtman, Konstantin Voronkov, Kris Zyp, Leif Hedstrom, Lionel Morand, Lisa Dusseault, Maciej Stachowiak, Manu Sporny, Marc Schneider, Marc Slemko, Mark Baker, Mark Pauley, Mark Watson, Markus Isomaki, Markus Lanthaler, Martin J. Duerst, Martin Musatov, Martin Nilsson, Martin Thomson, Matt Lynch, Matthew Cox, Matthew Kerwin, Max Clark, Menachem Dodge, Meral Shirazipour, Michael Burrows, Michael Hausenblas, Michael Scharf, Michael Sweet, Michael Tuexen, Michael Welzl, Mike Amundsen, Mike Belshe, Mike Bishop, Mike Kelly, Mike Schinkel, Miles Sabin, Murray S. Kucherawy, Mykyta Yevstifeyev, Nathan Rixham, Nicholas Shanks, Nico Williams, Nicolas Alvarez, Nicolas Mailhot, Noah Slater, Osama Mazahir, Pablo Castro, Pat Hayes, Patrick R. McManus, Paul E. Jones, Paul Hoffman, Paul Marquess, Pete Resnick, Peter Lepeska, Peter Occil, Peter Saint-Andre, Peter Watkins, Phil Archer, Phil Hunt, Philippe Mouglin, Phillip Hallam-Baker, Piotr Dobrogost, Poul-Henning Kamp, Preethi Natarajan, Rajeev Bector, Ray Polk, Reto Bachmann-Gmuer, Richard Barnes, Richard Cyganiak, Rob Trace, Robby Simpson, Robert Brewer, Robert Collins, Robert Mattson, Robert O'Callahan, Robert Olofsson, Robert Sayre, Robert Siemer, Robert de Wilde, Roberto Javier Godoy, Roberto Peon, Roland Zink, Ronny Widjaja, Ryan Hamilton, S. Mike Dierken, Salvatore Loreto, Sam Johnston, Sam Pullara, Sam Ruby, Saurabh Kulkarni, Scott Lawrence (who maintained the original issues list), Sean B. Palmer, Sean Turner, Sebastien Barnoud, Shane McCarron, Shigeki Ohtsu, Simon Yarde, Stefan Eissing, Stefan Tilkov, Stefanos Harhalakis, Stephane Bortzmeyer, Stephen Farrell, Stephen Kent, Stephen Ludin, Stuart Williams, Subbu Allamaraju, Subramanian Moonesamy, Susan Hares, Sylvain Hellegouarch, Tapan Divekar, Tatsuhiro Tsujikawa, Tatsuya Hayashi, Ted Hardie, Ted Lemon, Thomas Broyer, Thomas Fossati, Thomas Maslen, Thomas Nadeau, Thomas Nordin, Thomas Roessler, Tim Bray, Tim Morgan, Tim Olsen, Tom Zhou, Travis Snoozy, Tyler Close, Vincent Murphy, Wenbo Zhu, Werner Baumann, Wilbur Streett, Wilfredo Sanchez Vega, William A. Rowe Jr., William Chan, Willy Tarreau, Xiaoshu Wang, Yaron Goland, Yngve Nysaeter Pettersen, Yoav Nir, Yogesh Bang, Yuchung Cheng, Yutaka Oiwa, Yves Lafon (long-time member of the editor team), Zed A. Shaw, and Zhong Yu.

See Section 16 of [RFC2616] for additional acknowledgements from prior revisions.

11. References

11.1. Normative References

[Part2] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content",

draft-ietf-httpbis-p2-semantics-26 (work in progress),
February 2014.

- [Part4] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests", draft-ietf-httpbis-p4-conditional-26 (work in progress), February 2014.
- [Part5] Fielding, R., Ed., Lafon, Y., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Range Requests", draft-ietf-httpbis-p5-range-26 (work in progress), February 2014.
- [Part6] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Caching", draft-ietf-httpbis-p6-cache-26 (work in progress), February 2014.
- [Part7] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Authentication", draft-ietf-httpbis-p7-auth-26 (work in progress), February 2014.
- [RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.
- [RFC1950] Deutsch, L. and J-L. Gailly, "ZLIB Compressed Data Format Specification version 3.3", RFC 1950, May 1996.
- [RFC1951] Deutsch, P., "DEFLATE Compressed Data Format Specification version 1.3", RFC 1951, May 1996.
- [RFC1952] Deutsch, P., Gailly, J-L., Adler, M., Deutsch, L., and G. Randers-Pehrson, "GZIP file format specification version 4.3", RFC 1952, May 1996.
- [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., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008.
- [USASCII] American National Standards Institute, "Coded Character

Set -- 7-bit American Standard Code for Information Interchange", ANSI X3.4, 1986.

[Welch] Welch, T., "A Technique for High Performance Data Compression", IEEE Computer 17(6), June 1984.

11.2. Informative References

- [BCP115] Hansen, T., Hardie, T., and L. Masinter, "Guidelines and Registration Procedures for New URI Schemes", BCP 115, RFC 4395, February 2006.
- [BCP13] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, January 2013.
- [BCP90] Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, September 2004.
- [Georgiev] Georgiev, M., Iyengar, S., Jana, S., Anubhai, R., Boneh, D., and V. Shmatikov, "The Most Dangerous Code in the World: Validating SSL Certificates in Non-browser Software", In Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12), pp. 38-49, October 2012, <<http://doi.acm.org/10.1145/2382196.2382204>>.
- [ISO-8859-1] International Organization for Standardization, "Information technology -- 8-bit single-byte coded graphic character sets -- Part 1: Latin alphabet No. 1", ISO/IEC 8859-1:1998, 1998.
- [Klein] Klein, A., "Divide and Conquer - HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics", March 2004, <http://packetstormsecurity.com/papers/general/whitepaper_httpresponse.pdf>.
- [Kri2001] Kristol, D., "HTTP Cookies: Standards, Privacy, and Politics", ACM Transactions on Internet Technology 1(2), November 2001, <<http://arxiv.org/abs/cs.SE/0105018>>.
- [Linhart] Linhart, C., Klein, A., Heled, R., and S. Orrin, "HTTP Request Smuggling", June 2005, <<http://www.watchfire.com/news/whitepapers.aspx>>.
- [RFC1919] Chatel, M., "Classical versus Transparent IP Proxies",

RFC 1919, March 1996.

- [RFC1945] Berners-Lee, T., Fielding, R., and H. Nielsen, "Hypertext Transfer Protocol -- HTTP/1.0", RFC 1945, May 1996.
- [RFC2045] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, November 1996.
- [RFC2047] Moore, K., "MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text", RFC 2047, November 1996.
- [RFC2068] Fielding, R., Gettys, J., Mogul, J., Nielsen, H., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2068, January 1997.
- [RFC2145] Mogul, J., Fielding, R., Gettys, J., and H. Nielsen, "Use and Interpretation of HTTP Version Numbers", RFC 2145, May 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.
- [RFC2817] Khare, R. and S. Lawrence, "Upgrading to TLS Within HTTP/1.1", RFC 2817, May 2000.
- [RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.
- [RFC3040] Cooper, I., Melve, I., and G. Tomlinson, "Internet Web Replication and Caching Taxonomy", RFC 3040, January 2001.
- [RFC4033] Arends, R., Austein, R., Larson, M., Massey, D., and S. Rose, "DNS Security Introduction and Requirements", RFC 4033, March 2005.
- [RFC4559] Jaganathan, K., Zhu, L., and J. Brezak, "SPNEGO-based Kerberos and NTLM HTTP Authentication in Microsoft Windows", RFC 4559, June 2006.
- [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.

[RFC5322] Resnick, P., "Internet Message Format", RFC 5322, October 2008.

[RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, April 2011.

[RFC6585] Nottingham, M. and R. Fielding, "Additional HTTP Status Codes", RFC 6585, April 2012.

Appendix A. HTTP Version History

HTTP has been in use since 1990. The first version, later referred to as HTTP/0.9, was a simple protocol for hypertext data transfer across the Internet, using only a single request method (GET) and no metadata. HTTP/1.0, as defined by [RFC1945], added a range of request methods and MIME-like messaging, allowing for metadata to be transferred and modifiers placed on the request/response semantics. However, HTTP/1.0 did not sufficiently take into consideration the effects of hierarchical proxies, caching, the need for persistent connections, or name-based virtual hosts. The proliferation of incompletely-implemented applications calling themselves "HTTP/1.0" further necessitated a protocol version change in order for two communicating applications to determine each other's true capabilities.

HTTP/1.1 remains compatible with HTTP/1.0 by including more stringent requirements that enable reliable implementations, adding only those features that can either be safely ignored by an HTTP/1.0 recipient or only sent when communicating with a party advertising conformance with HTTP/1.1.

HTTP/1.1 has been designed to make supporting previous versions easy. A general-purpose HTTP/1.1 server ought to be able to understand any valid request in the format of HTTP/1.0, responding appropriately with an HTTP/1.1 message that only uses features understood (or safely ignored) by HTTP/1.0 clients. Likewise, an HTTP/1.1 client can be expected to understand any valid HTTP/1.0 response.

Since HTTP/0.9 did not support header fields in a request, there is no mechanism for it to support name-based virtual hosts (selection of resource by inspection of the Host header field). Any server that implements name-based virtual hosts ought to disable support for HTTP/0.9. Most requests that appear to be HTTP/0.9 are, in fact, badly constructed HTTP/1.x requests caused by a client failing to properly encode the request-target.

A.1. Changes from HTTP/1.0

This section summarizes major differences between versions HTTP/1.0 and HTTP/1.1.

A.1.1. Multi-homed Web Servers

The requirements that clients and servers support the Host header field (Section 5.4), report an error if it is missing from an HTTP/1.1 request, and accept absolute URIs (Section 5.3) are among the most important changes defined by HTTP/1.1.

Older HTTP/1.0 clients assumed a one-to-one relationship of IP addresses and servers; there was no other established mechanism for distinguishing the intended server of a request than the IP address to which that request was directed. The Host header field was introduced during the development of HTTP/1.1 and, though it was quickly implemented by most HTTP/1.0 browsers, additional requirements were placed on all HTTP/1.1 requests in order to ensure complete adoption. At the time of this writing, most HTTP-based services are dependent upon the Host header field for targeting requests.

A.1.2. Keep-Alive Connections

In HTTP/1.0, each connection is established by the client prior to the request and closed by the server after sending the response. However, some implementations implement the explicitly negotiated ("Keep-Alive") version of persistent connections described in Section 19.7.1 of [RFC2068].

Some clients and servers might wish to be compatible with these previous approaches to persistent connections, by explicitly negotiating for them with a "Connection: keep-alive" request header field. However, some experimental implementations of HTTP/1.0 persistent connections are faulty; for example, if an HTTP/1.0 proxy server doesn't understand Connection, it will erroneously forward that header field to the next inbound server, which would result in a hung connection.

One attempted solution was the introduction of a Proxy-Connection header field, targeted specifically at proxies. In practice, this was also unworkable, because proxies are often deployed in multiple layers, bringing about the same problem discussed above.

As a result, clients are encouraged not to send the Proxy-Connection header field in any requests.

Clients are also encouraged to consider the use of Connection: keep-alive in requests carefully; while they can enable persistent connections with HTTP/1.0 servers, clients using them will need to monitor the connection for "hung" requests (which indicate that the client ought stop sending the header field), and this mechanism ought not be used by clients at all when a proxy is being used.

A.1.3. Introduction of Transfer-Encoding

HTTP/1.1 introduces the Transfer-Encoding header field (Section 3.3.1). Transfer codings need to be decoded prior to forwarding an HTTP message over a MIME-compliant protocol.

A.2. Changes from RFC 2616

HTTP's approach to error handling has been explained. (Section 2.5)

The HTTP-version ABNF production has been clarified to be case-sensitive. Additionally, version numbers has been restricted to single digits, due to the fact that implementations are known to handle multi-digit version numbers incorrectly. (Section 2.6)

Userinfo (i.e., username and password) are now disallowed in HTTP and HTTPS URIs, because of security issues related to their transmission on the wire. (Section 2.7.1)

The HTTPS URI scheme is now defined by this specification; previously, it was done in Section 2.4 of [RFC2818]. Furthermore, it implies end-to-end security. (Section 2.7.2)

HTTP messages can be (and often are) buffered by implementations; despite it sometimes being available as a stream, HTTP is fundamentally a message-oriented protocol. Minimum supported sizes for various protocol elements have been suggested, to improve interoperability. (Section 3)

Invalid whitespace around field-names is now required to be rejected, because accepting it represents a security vulnerability. The ABNF productions defining header fields now only list the field value. (Section 3.2)

Rules about implicit linear whitespace between certain grammar productions have been removed; now whitespace is only allowed where specifically defined in the ABNF. (Section 3.2.3)

Header fields that span multiple lines ("line folding") are deprecated. (Section 3.2.4)

The NUL octet is no longer allowed in comment and quoted-string text, and handling of backslash-escaping in them has been clarified. The quoted-pair rule no longer allows escaping control characters other than HTAB. Non-ASCII content in header fields and the reason phrase has been obsoleted and made opaque (the TEXT rule was removed). (Section 3.2.6)

Bogus "Content-Length" header fields are now required to be handled as errors by recipients. (Section 3.3.2)

The algorithm for determining the message body length has been clarified to indicate all of the special cases (e.g., driven by methods or status codes) that affect it, and that new protocol elements cannot define such special cases. CONNECT is a new, special case in determining message body length. "multipart/byteranges" is no longer a way of determining message body length detection. (Section 3.3.3)

The "identity" transfer coding token has been removed. (Sections 3.3 and 4)

Chunk length does not include the count of the octets in the chunk header and trailer. Line folding in chunk extensions is disallowed. (Section 4.1)

The meaning of the "deflate" content coding has been clarified. (Section 4.2.2)

The segment + query components of RFC 3986 have been used to define the request-target, instead of `abs_path` from RFC 1808. The asterisk-form of the request-target is only allowed with the OPTIONS method. (Section 5.3)

The term "Effective Request URI" has been introduced. (Section 5.5)

Gateways do not need to generate Via header fields anymore. (Section 5.7.1)

Exactly when "close" connection options have to be sent has been clarified. Also, "hop-by-hop" header fields are required to appear in the Connection header field; just because they're defined as hop-by-hop in this specification doesn't exempt them. (Section 6.1)

The limit of two connections per server has been removed. An idempotent sequence of requests is no longer required to be retried. The requirement to retry requests under certain circumstances when the server prematurely closes the connection has been removed. Also, some extraneous requirements about when servers are allowed to close

connections prematurely have been removed. (Section 6.3)

The semantics of the Upgrade header field is now defined in responses other than 101 (this was incorporated from [RFC2817]). Furthermore, the ordering in the field value is now significant. (Section 6.7)

Empty list elements in list productions (e.g., a list header field containing ", ,") have been deprecated. (Section 7)

Registration of Transfer Codings now requires IETF Review (Section 8.4)

This specification now defines the Upgrade Token Registry, previously defined in Section 7.2 of [RFC2817]. (Section 8.6)

The expectation to support HTTP/0.9 requests has been removed. (Appendix A)

Issues with the Keep-Alive and Proxy-Connection header fields in requests are pointed out, with use of the latter being discouraged altogether. (Appendix A.1.2)

Appendix B. Collected ABNF

BWS = OWS

Connection = *("," OWS) connection-option *(OWS "," [OWS
connection-option])
Content-Length = 1*DIGIT

HTTP-message = start-line *(header-field CRLF) CRLF [message-body
]

HTTP-name = %x48.54.54.50 ; HTTP
HTTP-version = HTTP-name "/" DIGIT "." DIGIT
Host = uri-host [":" port]

OWS = *(SP / HTAB)

RWS = 1*(SP / HTAB)

TE = [("," / t-codings) *(OWS "," [OWS t-codings])]
Trailer = *("," OWS) field-name *(OWS "," [OWS field-name])
Transfer-Encoding = *("," OWS) transfer-coding *(OWS "," [OWS
transfer-coding])

URI-reference = <URI-reference, defined in [RFC3986], Section 4.1>
Upgrade = *("," OWS) protocol *(OWS "," [OWS protocol])

```
Via = *( "," OWS ) ( received-protocol RWS received-by [ RWS comment
  ] ) *( OWS "," [ OWS ( received-protocol RWS received-by [ RWS
  comment ] ) ] )
```

```
absolute-URI = <absolute-URI, defined in [RFC3986], Section 4.3>
absolute-form = absolute-URI
absolute-path = 1*( "/" segment )
asterisk-form = "*"
authority = <authority, defined in [RFC3986], Section 3.2>
authority-form = authority
```

```
chunk = chunk-size [ chunk-ext ] CRLF chunk-data CRLF
chunk-data = 1*OCTET
chunk-ext = *( ";" chunk-ext-name [ "=" chunk-ext-val ] )
chunk-ext-name = token
chunk-ext-val = token / quoted-string
chunk-size = 1*HEXDIG
chunked-body = *chunk last-chunk trailer-part CRLF
comment = "(" *( ctext / quoted-pair / comment ) ")"
connection-option = token
ctext = HTAB / SP / %x21-27 ; '!'-'~'
      / %x2A-5B ; '*'-'['
      / %x5D-7E ; ']'-'~'
      / obs-text
```

```
field-content = field-vchar [ 1*( SP / HTAB ) field-vchar ]
field-name = token
field-value = *( field-content / obs-fold )
field-vchar = VCHAR / obs-text
fragment = <fragment, defined in [RFC3986], Section 3.5>
```

```
header-field = field-name ":" OWS field-value OWS
http-URI = "http://" authority path-abempty [ "?" query ] [ "#"
  fragment ]
https-URI = "https://" authority path-abempty [ "?" query ] [ "#"
  fragment ]
```

```
last-chunk = 1*"0" [ chunk-ext ] CRLF
```

```
message-body = *OCTET
method = token
```

```
obs-fold = CRLF 1*( SP / HTAB )
obs-text = %x80-FF
origin-form = absolute-path [ "?" query ]
```

```
partial-URI = relative-part [ "?" query ]
path-abempty = <path-abempty, defined in [RFC3986], Section 3.3>
```

```

port = <port, defined in [RFC3986], Section 3.2.3>
protocol = protocol-name [ "/" protocol-version ]
protocol-name = token
protocol-version = token
pseudonym = token

qdtexT = HTAB / SP / "!" / %x23-5B ; '#'-'['
        / %x5D-7E ; ']'-'~'
        / obs-text
query = <query, defined in [RFC3986], Section 3.4>
quoted-pair = "\" ( HTAB / SP / VCHAR / obs-text )
quoted-string = DQUOTE *( qdtexT / quoted-pair ) DQUOTE

rank = ( "0" [ "." *3DIGIT ] ) / ( "1" [ "." *3"0" ] )
reason-phrase = *( HTAB / SP / VCHAR / obs-text )
received-by = ( uri-host [ ":" port ] ) / pseudonym
received-protocol = [ protocol-name "/" ] protocol-version
relative-part = <relative-part, defined in [RFC3986], Section 4.2>
request-line = method SP request-target SP HTTP-version CRLF
request-target = origin-form / absolute-form / authority-form /
        asterisk-form

scheme = <scheme, defined in [RFC3986], Section 3.1>
segment = <segment, defined in [RFC3986], Section 3.3>
start-line = request-line / status-line
status-code = 3DIGIT
status-line = HTTP-version SP status-code SP reason-phrase CRLF

t-codings = "trailers" / ( transfer-coding [ t-ranking ] )
t-ranking = OWS ";" OWS "q=" rank
tchar = "!" / "#" / "$" / "%" / "&" / "'" / "*" / "+" / "-" / "." /
        "^" / "_" / "`" / "|" / "~" / DIGIT / ALPHA
token = 1*tchar
trailer-part = *( header-field CRLF )
transfer-coding = "chunked" / "compress" / "deflate" / "gzip" /
        transfer-extension
transfer-extension = token *( OWS ";" OWS transfer-parameter )
transfer-parameter = token BWS "=" BWS ( token / quoted-string )

uri-host = <host, defined in [RFC3986], Section 3.2.2>

```

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

C.1. Since RFC 2616

Changes up to the IETF Last Call draft are summarized in <<http://trac.tools.ietf.org/html/draft-ietf-httpbis-pl-messaging-24#appendix-C>>.

C.2. Since draft-ietf-httpbis-pl-messaging-24

Closed issues:

- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/502>>: "APPSDIR review of draft-ietf-httpbis-pl-messaging-24"
- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/507>>: "integer value parsing"
- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/517>>: "move IANA registrations to correct draft"

C.3. Since draft-ietf-httpbis-pl-messaging-25

Closed issues:

- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/526>>: "check media type registration templates"
- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/528>>: "Redundant rule quoted-str-nf"
- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/531>>: "IESG ballot on draft-ietf-httpbis-pl-messaging-25"
- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/538>>: "add 'stateless' to Abstract"
- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/540>>: "clarify ABNF layering"
- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/541>>: "use of 'word' ABNF production"
- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/542>>: "improve introduction of list rule"
- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/544>>: "moving 2616/2068/2145 to historic"
- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/549>>: "augment security considerations with pointers to current research"
- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/551>>: "intermediaries handling trailers"

- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/552>>: "allow privacy proxies to be conformant"

Index

A

absolute-form (of request-target) 42
accelerator 10
application/http Media Type 62
asterisk-form (of request-target) 42
authoritative response 66
authority-form (of request-target) 42

B

browser 7

C

cache 11
cacheable 11
captive portal 11
chunked (Coding Format) 28, 31, 35
client 7
close 50, 55
compress (Coding Format) 38
connection 7
Connection header field 50, 55
Content-Length header field 30

D

deflate (Coding Format) 38
Delimiters 26
downstream 9

E

effective request URI 44

G

gateway 10
Grammar
 absolute-form 41-42
 absolute-path 16
 absolute-URI 16
 ALPHA 6
 asterisk-form 41-42
 authority 16
 authority-form 41-42
 BWS 24
 chunk 35

chunk-data 35
chunk-ext 35-36
chunk-ext-name 36
chunk-ext-val 36
chunk-size 35
chunked-body 35-36
comment 27
Connection 51
connection-option 51
Content-Length 30
CR 6
CRLF 6
ctext 27
CTL 6
DIGIT 6
DQUOTE 6
field-content 22
field-name 22, 39
field-value 22
field-vchar 22
fragment 16
header-field 22, 36
HEXDIG 6
Host 43
HTAB 6
HTTP-message 19
HTTP-name 13
http-URI 16
HTTP-version 13
https-URI 18
last-chunk 35
LF 6
message-body 27
method 21
obs-fold 22
obs-text 27
OCTET 6
origin-form 41
OWS 24
partial-URI 16
port 16
protocol-name 47
protocol-version 47
pseudonym 47
qdtex 27
query 16
quoted-pair 27
quoted-string 27

- rank 38
- reason-phrase 22
- received-by 47
- received-protocol 47
- request-line 21
- request-target 41
- RWS 24
- scheme 16
- segment 16
- SP 6
- start-line 20
- status-code 22
- status-line 22
- t-codings 38
- t-ranking 38
- tchar 27
- TE 38
- token 27
- Trailer 39
- trailer-part 35-36
- transfer-coding 35
- Transfer-Encoding 28
- transfer-extension 35
- transfer-parameter 35
- Upgrade 56
- uri-host 16
- URI-reference 16
- VCHAR 6
- Via 47
- gzip (Coding Format) 38

H

- header field 19
- header section 19
- headers 19
- Host header field 43
- http URI scheme 16
- https URI scheme 18

I

- inbound 9
- interception proxy 11
- intermediary 9

M

- Media Type
 - application/http 62
 - message/http 61

message 7
message/http Media Type 61
method 21

N
non-transforming proxy 48

O
origin server 7
origin-form (of request-target) 41
outbound 9

P
phishing 66
proxy 10

R
recipient 7
request 7
request-target 21
resource 16
response 7
reverse proxy 10

S
sender 7
server 7
spider 7

T
target resource 40
target URI 40
TE header field 38
Trailer header field 39
Transfer-Encoding header field 28
transforming proxy 48
transparent proxy 11
tunnel 10

U
Upgrade header field 56
upstream 9
URI scheme
 http 16
 https 18
user agent 7

V

Via header field 47

Authors' Addresses

Roy T. Fielding (editor)
Adobe Systems Incorporated
345 Park Ave
San Jose, CA 95110
USA

EMail: fielding@gbiv.com
URI: <http://roy.gbiv.com/>

Julian F. Reschke (editor)
greenbytes GmbH
Hafenweg 16
Muenster, NW 48155
Germany

EMail: julian.reschke@greenbytes.de
URI: <http://greenbytes.de/tech/webdav/>

HTTPbis Working Group
Internet-Draft
Obsoletes: 2616 (if approved)
Updates: 2817 (if approved)
Intended status: Standards Track
Expires: August 10, 2014

R. Fielding, Ed.
Adobe
J. Reschke, Ed.
greenbytes
February 6, 2014

Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content
draft-ietf-httpbis-p2-semantics-26

Abstract

The Hypertext Transfer Protocol (HTTP) is a stateless application-level protocol for distributed, collaborative, hypertext information systems. This document defines the semantics of HTTP/1.1 messages, as expressed by request methods, request header fields, response status codes, and response header fields, along with the payload of messages (metadata and body content) and mechanisms for content negotiation.

Editorial Note (To be removed by RFC Editor)

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

The current issues list is at <http://tools.ietf.org/wg/httpbis/trac/report/3> and related documents (including fancy diffs) can be found at <http://tools.ietf.org/wg/httpbis/>.

The changes in this draft are summarized in Appendix E.3.

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 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.

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.

Table of Contents

| | |
|--|----|
| 1. Introduction | 6 |
| 1.1. Conformance and Error Handling | 6 |
| 1.2. Syntax Notation | 6 |
| 2. Resources | 7 |
| 3. Representations | 7 |
| 3.1. Representation Metadata | 8 |
| 3.1.1. Processing Representation Data | 8 |
| 3.1.2. Encoding for Compression or Integrity | 11 |
| 3.1.3. Audience Language | 13 |
| 3.1.4. Identification | 14 |
| 3.2. Representation Data | 17 |
| 3.3. Payload Semantics | 17 |
| 3.4. Content Negotiation | 18 |
| 3.4.1. Proactive Negotiation | 19 |
| 3.4.2. Reactive Negotiation | 20 |
| 4. Request Methods | 21 |
| 4.1. Overview | 21 |

| | | |
|--------|---|----|
| 4.2. | Common Method Properties | 22 |
| 4.2.1. | Safe Methods | 22 |
| 4.2.2. | Idempotent Methods | 23 |
| 4.2.3. | Cacheable Methods | 24 |
| 4.3. | Method Definitions | 24 |
| 4.3.1. | GET | 24 |
| 4.3.2. | HEAD | 25 |
| 4.3.3. | POST | 25 |
| 4.3.4. | PUT | 26 |
| 4.3.5. | DELETE | 29 |
| 4.3.6. | CONNECT | 30 |
| 4.3.7. | OPTIONS | 31 |
| 4.3.8. | TRACE | 32 |
| 5. | Request Header Fields | 33 |
| 5.1. | Controls | 33 |
| 5.1.1. | Expect | 33 |
| 5.1.2. | Max-Forwards | 36 |
| 5.2. | Conditionals | 36 |
| 5.3. | Content Negotiation | 37 |
| 5.3.1. | Quality Values | 37 |
| 5.3.2. | Accept | 38 |
| 5.3.3. | Accept-Charset | 40 |
| 5.3.4. | Accept-Encoding | 41 |
| 5.3.5. | Accept-Language | 42 |
| 5.4. | Authentication Credentials | 43 |
| 5.5. | Request Context | 44 |
| 5.5.1. | From | 44 |
| 5.5.2. | Referer | 44 |
| 5.5.3. | User-Agent | 46 |
| 6. | Response Status Codes | 47 |
| 6.1. | Overview of Status Codes | 47 |
| 6.2. | Informational 1xx | 49 |
| 6.2.1. | 100 Continue | 49 |
| 6.2.2. | 101 Switching Protocols | 49 |
| 6.3. | Successful 2xx | 50 |
| 6.3.1. | 200 OK | 50 |
| 6.3.2. | 201 Created | 51 |
| 6.3.3. | 202 Accepted | 51 |
| 6.3.4. | 203 Non-Authoritative Information | 51 |
| 6.3.5. | 204 No Content | 52 |
| 6.3.6. | 205 Reset Content | 52 |
| 6.4. | Redirection 3xx | 53 |
| 6.4.1. | 300 Multiple Choices | 54 |
| 6.4.2. | 301 Moved Permanently | 55 |
| 6.4.3. | 302 Found | 55 |
| 6.4.4. | 303 See Other | 56 |
| 6.4.5. | 305 Use Proxy | 56 |
| 6.4.6. | 306 (Unused) | 56 |

| | | |
|---------|--|----|
| 6.4.7. | 307 Temporary Redirect | 57 |
| 6.5. | Client Error 4xx | 57 |
| 6.5.1. | 400 Bad Request | 57 |
| 6.5.2. | 402 Payment Required | 57 |
| 6.5.3. | 403 Forbidden | 57 |
| 6.5.4. | 404 Not Found | 58 |
| 6.5.5. | 405 Method Not Allowed | 58 |
| 6.5.6. | 406 Not Acceptable | 58 |
| 6.5.7. | 408 Request Timeout | 59 |
| 6.5.8. | 409 Conflict | 59 |
| 6.5.9. | 410 Gone | 59 |
| 6.5.10. | 411 Length Required | 60 |
| 6.5.11. | 413 Payload Too Large | 60 |
| 6.5.12. | 414 URI Too Long | 60 |
| 6.5.13. | 415 Unsupported Media Type | 60 |
| 6.5.14. | 417 Expectation Failed | 61 |
| 6.5.15. | 426 Upgrade Required | 61 |
| 6.6. | Server Error 5xx | 61 |
| 6.6.1. | 500 Internal Server Error | 61 |
| 6.6.2. | 501 Not Implemented | 62 |
| 6.6.3. | 502 Bad Gateway | 62 |
| 6.6.4. | 503 Service Unavailable | 62 |
| 6.6.5. | 504 Gateway Timeout | 62 |
| 6.6.6. | 505 HTTP Version Not Supported | 62 |
| 7. | Response Header Fields | 63 |
| 7.1. | Control Data | 63 |
| 7.1.1. | Origination Date | 63 |
| 7.1.2. | Location | 67 |
| 7.1.3. | Retry-After | 68 |
| 7.1.4. | Vary | 69 |
| 7.2. | Validator Header Fields | 70 |
| 7.3. | Authentication Challenges | 71 |
| 7.4. | Response Context | 71 |
| 7.4.1. | Allow | 71 |
| 7.4.2. | Server | 72 |
| 8. | IANA Considerations | 72 |
| 8.1. | Method Registry | 73 |
| 8.1.1. | Procedure | 73 |
| 8.1.2. | Considerations for New Methods | 73 |
| 8.1.3. | Registrations | 74 |
| 8.2. | Status Code Registry | 74 |
| 8.2.1. | Procedure | 74 |
| 8.2.2. | Considerations for New Status Codes | 75 |
| 8.2.3. | Registrations | 75 |
| 8.3. | Header Field Registry | 76 |
| 8.3.1. | Considerations for New Header Fields | 77 |
| 8.3.2. | Registrations | 79 |
| 8.4. | Content Coding Registry | 79 |

| | | |
|-------------|--|----|
| 8.4.1. | Procedure | 80 |
| 8.4.2. | Registrations | 80 |
| 9. | Security Considerations | 80 |
| 9.1. | Attacks Based On File and Path Names | 81 |
| 9.2. | Attacks Based On Command, Code, or Query Injection | 81 |
| 9.3. | Disclosure of Personal Information | 82 |
| 9.4. | Disclosure of Sensitive Information in URIs | 82 |
| 9.5. | Disclosure of Fragment after Redirects | 82 |
| 9.6. | Disclosure of Product Information | 83 |
| 9.7. | Browser Fingerprinting | 83 |
| 10. | Acknowledgments | 84 |
| 11. | References | 84 |
| 11.1. | Normative References | 84 |
| 11.2. | Informative References | 85 |
| Appendix A. | Differences between HTTP and MIME | 87 |
| A.1. | MIME-Version | 88 |
| A.2. | Conversion to Canonical Form | 88 |
| A.3. | Conversion of Date Formats | 88 |
| A.4. | Conversion of Content-Encoding | 89 |
| A.5. | Conversion of Content-Transfer-Encoding | 89 |
| A.6. | MHTML and Line Length Limitations | 89 |
| Appendix B. | Changes from RFC 2616 | 89 |
| Appendix C. | Imported ABNF | 92 |
| Appendix D. | Collected ABNF | 92 |
| Appendix E. | Change Log (to be removed by RFC Editor before publication) | 95 |
| E.1. | Since RFC 2616 | 95 |
| E.2. | Since draft-ietf-httpbis-p2-semantics-24 | 95 |
| E.3. | Since draft-ietf-httpbis-p2-semantics-25 | 96 |
| Index | | 96 |

1. Introduction

Each Hypertext Transfer Protocol (HTTP) message is either a request or a response. A server listens on a connection for a request, parses each message received, interprets the message semantics in relation to the identified request target, and responds to that request with one or more response messages. A client constructs request messages to communicate specific intentions, and examines received responses to see if the intentions were carried out and determine how to interpret the results. This document defines HTTP/1.1 request and response semantics in terms of the architecture defined in [Part1].

HTTP provides a uniform interface for interacting with a resource (Section 2), regardless of its type, nature, or implementation, via the manipulation and transfer of representations (Section 3).

HTTP semantics include the intentions defined by each request method (Section 4), extensions to those semantics that might be described in request header fields (Section 5), the meaning of status codes to indicate a machine-readable response (Section 6), and the meaning of other control data and resource metadata that might be given in response header fields (Section 7).

This document also defines representation metadata that describe how a payload is intended to be interpreted by a recipient, the request header fields that might influence content selection, and the various selection algorithms that are collectively referred to as "content negotiation" (Section 3.4).

1.1. Conformance and Error Handling

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].

Conformance criteria and considerations regarding error handling are defined in Section 2.5 of [Part1].

1.2. Syntax Notation

This specification uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234] with a list extension, defined in Section 7 of [Part1], that allows for compact definition of comma-separated lists using a '#' operator (similar to how the '*' operator indicates repetition). Appendix C describes rules imported from other documents. Appendix D shows the collected grammar with all list operators expanded to standard ABNF notation.

This specification uses the terms "character", "character encoding scheme", "charset", and "protocol element" as they are defined in [RFC6365].

2. Resources

The target of an HTTP request is called a resource. HTTP does not limit the nature of a resource; it merely defines an interface that might be used to interact with resources. Each resource is identified by a Uniform Resource Identifier (URI), as described in Section 2.7 of [Part1].

When a client constructs an HTTP/1.1 request message, it sends the target URI in one of various forms, as defined in (Section 5.3 of [Part1]). When a request is received, the server reconstructs an effective request URI for the target resource (Section 5.5 of [Part1]).

One design goal of HTTP is to separate resource identification from request semantics, which is made possible by vesting the request semantics in the request method (Section 4) and a few request-modifying header fields (Section 5). If there is a conflict between the method semantics and any semantic implied by the URI itself, as described in Section 4.2.1, the method semantics take precedence.

3. Representations

Considering that a resource could be anything, and that the uniform interface provided by HTTP is similar to a window through which one can observe and act upon such a thing only through the communication of messages to some independent actor on the other side, an abstraction is needed to represent ("take the place of") the current or desired state of that thing in our communications. That abstraction is called a representation [REST].

For the purposes of HTTP, a "representation" is information that is intended to reflect a past, current, or desired state of a given resource, in a format that can be readily communicated via the protocol, and that consists of a set of representation metadata and a potentially unbounded stream of representation data.

An origin server might be provided with, or capable of generating, multiple representations that are each intended to reflect the current state of a target resource. In such cases, some algorithm is used by the origin server to select one of those representations as most applicable to a given request, usually based on content negotiation. This "selected representation" is used to provide the data and metadata for evaluating conditional requests [Part4] and

constructing the payload for 200 (OK) and 304 (Not Modified) responses to GET (Section 4.3.1).

3.1. Representation Metadata

Representation header fields provide metadata about the representation. When a message includes a payload body, the representation header fields describe how to interpret the representation data enclosed in the payload body. In a response to a HEAD request, the representation header fields describe the representation data that would have been enclosed in the payload body if the same request had been a GET.

The following header fields convey representation metadata:

| Header Field Name | Defined in... |
|-------------------|-----------------|
| Content-Type | Section 3.1.1.5 |
| Content-Encoding | Section 3.1.2.2 |
| Content-Language | Section 3.1.3.2 |
| Content-Location | Section 3.1.4.2 |

3.1.1. Processing Representation Data

3.1.1.1. Media Type

HTTP uses Internet Media Types [RFC2046] in the Content-Type (Section 3.1.1.5) and Accept (Section 5.3.2) header fields in order to provide open and extensible data typing and type negotiation. Media types define both a data format and various processing models: how to process that data in accordance with each context in which it is received.

```
media-type = type "/" subtype *( OWS ";" OWS parameter )
type       = token
subtype    = token
```

The type/subtype MAY be followed by parameters in the form of name=value pairs.

```
parameter = token "=" ( token / quoted-string )
```

The type, subtype, and parameter name tokens are case-insensitive. Parameter values might or might not be case-sensitive, depending on the semantics of the parameter name. The presence or absence of a parameter might be significant to the processing of a media-type,

depending on its definition within the media type registry.

A parameter value that matches the token production can be transmitted as either a token or within a quoted-string. The quoted and unquoted values are equivalent. For example, the following examples are all equivalent, but the first is preferred for consistency:

```
text/html;charset=utf-8
text/html;charset=UTF-8
Text/HTML;Charset="utf-8"
text/html; charset="utf-8"
```

Internet media types ought to be registered with IANA according to the procedures defined in [BCP13].

Note: Unlike some similar constructs in other header fields, media type parameters do not allow whitespace (even "bad" whitespace) around the "=" character.

3.1.1.2. Charset

HTTP uses charset names to indicate or negotiate the character encoding scheme of a textual representation [RFC6365]. A charset is identified by a case-insensitive token.

charset = token

Charset names ought to be registered in IANA Character Set registry (<<http://www.iana.org/assignments/character-sets>>) according to the procedures defined in [RFC2978].

3.1.1.3. Canonicalization and Text Defaults

Internet media types are registered with a canonical form in order to be interoperable among systems with varying native encoding formats. Representations selected or transferred via HTTP ought to be in canonical form, for many of the same reasons described by the Multipurpose Internet Mail Extensions (MIME) [RFC2045]. However, the performance characteristics of email deployments (i.e., store and forward messages to peers) are significantly different from those common to HTTP and the Web (server-based information services). Furthermore, MIME's constraints for the sake of compatibility with older mail transfer protocols do not apply to HTTP (see Appendix A).

MIME's canonical form requires that media subtypes of the "text" type use CRLF as the text line break. HTTP allows the transfer of text media with plain CR or LF alone representing a line break, when such

line breaks are consistent for an entire representation. An HTTP sender MAY generate, and a recipient MUST be able to parse, line breaks in text media that consist of CRLF, bare CR, or bare LF. In addition, text media in HTTP is not limited to charsets that use octets 13 and 10 for CR and LF, respectively. This flexibility regarding line breaks applies only to text within a representation that has been assigned a "text" media type; it does not apply to "multipart" types or HTTP elements outside the payload body (e.g., header fields).

If a representation is encoded with a content-coding, the underlying data ought to be in a form defined above prior to being encoded.

3.1.1.4. Multipart Types

MIME provides for a number of "multipart" types -- encapsulations of one or more representations within a single message body. All multipart types share a common syntax, as defined in Section 5.1.1 of [RFC2046], and include a boundary parameter as part of the media type value. The message body is itself a protocol element; a sender MUST generate only CRLF to represent line breaks between body parts.

HTTP message framing does not use the multipart boundary as an indicator of message body length, though it might be used by implementations that generate or process the payload. For example, the "multipart/form-data" type is often used for carrying form data in a request, as described in [RFC2388], and the "multipart/byteranges" type is defined by this specification for use in some 206 (Partial Content) responses [Part5].

3.1.1.5. Content-Type

The "Content-Type" header field indicates the media type of the associated representation: either the representation enclosed in the message payload or the selected representation, as determined by the message semantics. The indicated media type defines both the data format and how that data is intended to be processed by a recipient, within the scope of the received message semantics, after any content codings indicated by Content-Encoding are decoded.

Content-Type = media-type

Media types are defined in Section 3.1.1.1. An example of the field is

Content-Type: text/html; charset=ISO-8859-4

A sender that generates a message containing a payload body SHOULD

generate a Content-Type header field in that message unless the intended media type of the enclosed representation is unknown to the sender. If a Content-Type header field is not present, the recipient MAY either assume a media type of "application/octet-stream" ([RFC2046], Section 4.5.1) or examine the data to determine its type.

In practice, resource owners do not always properly configure their origin server to provide the correct Content-Type for a given representation, with the result that some clients will examine a payload's content and override the specified type. Clients that do so risk drawing incorrect conclusions, which might expose additional security risks (e.g., "privilege escalation"). Furthermore, it is impossible to determine the sender's intent by examining the data format: many data formats match multiple media types that differ only in processing semantics. Implementers are encouraged to provide a means of disabling such "content sniffing" when it is used.

3.1.2. Encoding for Compression or Integrity

3.1.2.1. Content Codings

Content coding values indicate an encoding transformation that has been or can be applied to a representation. Content codings are primarily used to allow a representation to be compressed or otherwise usefully transformed without losing the identity of its underlying media type and without loss of information. Frequently, the representation is stored in coded form, transmitted directly, and only decoded by the final recipient.

content-coding = token

All content-coding values are case-insensitive and ought to be registered within the HTTP Content Coding registry, as defined in Section 8.4. They are used in the Accept-Encoding (Section 5.3.4) and Content-Encoding (Section 3.1.2.2) header fields.

The following content-coding values are defined by this specification:

compress (and x-compress): See Section 4.2.1 of [Part1].

deflate: See Section 4.2.2 of [Part1].

gzip (and x-gzip): See Section 4.2.3 of [Part1].

3.1.2.2. Content-Encoding

The "Content-Encoding" header field indicates what content codings have been applied to the representation, beyond those inherent in the media type, and thus what decoding mechanisms have to be applied in order to obtain data in the media type referenced by the Content-Type header field. Content-Encoding is primarily used to allow a representation's data to be compressed without losing the identity of its underlying media type.

Content-Encoding = 1#content-coding

An example of its use is

Content-Encoding: gzip

If one or more encodings have been applied to a representation, the sender that applied the encodings **MUST** generate a Content-Encoding header field that lists the content codings in the order in which they were applied. Additional information about the encoding parameters can be provided by other header fields not defined by this specification.

Unlike Transfer-Encoding (Section 3.3.1 of [Part1]), the codings listed in Content-Encoding are a characteristic of the representation; the representation is defined in terms of the coded form, and all other metadata about the representation is about the coded form unless otherwise noted in the metadata definition. Typically, the representation is only decoded just prior to rendering or analogous usage.

If the media type includes an inherent encoding, such as a data format that is always compressed, then that encoding would not be restated in Content-Encoding even if it happens to be the same algorithm as one of the content codings. Such a content coding would only be listed if, for some bizarre reason, it is applied a second time to form the representation. Likewise, an origin server might choose to publish the same data as multiple representations that differ only in whether the coding is defined as part of Content-Type or Content-Encoding, since some user agents will behave differently in their handling of each response (e.g., open a "Save as ..." dialog instead of automatic decompression and rendering of content).

An origin server **MAY** respond with a status code of 415 (Unsupported Media Type) if a representation in the request message has a content coding that is not acceptable.

3.1.3. Audience Language

3.1.3.1. Language Tags

A language tag, as defined in [RFC5646], identifies a natural language spoken, written, or otherwise conveyed by human beings for communication of information to other human beings. Computer languages are explicitly excluded.

HTTP uses language tags within the Accept-Language and Content-Language header fields. Accept-Language uses the broader language-range production defined in Section 5.3.5, whereas Content-Language uses the language-tag production defined below.

language-tag = <Language-Tag, defined in [RFC5646], Section 2.1>

A language tag is a sequence of one or more case-insensitive subtags, each separated by a hyphen character ("-", %x2D). In most cases, a language tag consists of a primary language subtag that identifies a broad family of related languages (e.g., "en" = English) which is optionally followed by a series of subtags that refine or narrow that language's range (e.g., "en-CA" = the variety of English as communicated in Canada). Whitespace is not allowed within a language tag. Example tags include:

fr, en-US, es-419, az-Arab, x-pig-latin, man-Nkoo-GN

See [RFC5646] for further information.

3.1.3.2. Content-Language

The "Content-Language" header field describes the natural language(s) of the intended audience for the representation. Note that this might not be equivalent to all the languages used within the representation.

Content-Language = 1#language-tag

Language tags are defined in Section 3.1.3.1. The primary purpose of Content-Language is to allow a user to identify and differentiate representations according to the users' own preferred language. Thus, if the content is intended only for a Danish-literate audience, the appropriate field is

Content-Language: da

If no Content-Language is specified, the default is that the content is intended for all language audiences. This might mean that the

sender does not consider it to be specific to any natural language, or that the sender does not know for which language it is intended.

Multiple languages MAY be listed for content that is intended for multiple audiences. For example, a rendition of the "Treaty of Waitangi", presented simultaneously in the original Maori and English versions, would call for

Content-Language: mi, en

However, just because multiple languages are present within a representation does not mean that it is intended for multiple linguistic audiences. An example would be a beginner's language primer, such as "A First Lesson in Latin", which is clearly intended to be used by an English-literate audience. In this case, the Content-Language would properly only include "en".

Content-Language MAY be applied to any media type -- it is not limited to textual documents.

3.1.4. Identification

3.1.4.1. Identifying a Representation

When a complete or partial representation is transferred in a message payload, it is often desirable for the sender to supply, or the recipient to determine, an identifier for a resource corresponding to that representation.

For a request message:

- o If the request has a Content-Location header field, then the sender asserts that the payload is a representation of the resource identified by the Content-Location field-value. However, such an assertion cannot be trusted unless it can be verified by other means (not defined by this specification). The information might still be useful for revision history links.
- o Otherwise, the payload is unidentified.

For a response message, the following rules are applied in order until a match is found:

1. If the request method is GET or HEAD and the response status code is 200 (OK), 204 (No Content), 206 (Partial Content), or 304 (Not Modified), the payload is a representation of the resource identified by the effective request URI (Section 5.5 of [Part1]).

2. If the request method is GET or HEAD and the response status code is 203 (Non-Authoritative Information), the payload is a potentially modified or enhanced representation of the target resource as provided by an intermediary.
3. If the response has a Content-Location header field and its field-value is a reference to the same URI as the effective request URI, the payload is a representation of the resource identified by the effective request URI.
4. If the response has a Content-Location header field and its field-value is a reference to a URI different from the effective request URI, then the sender asserts that the payload is a representation of the resource identified by the Content-Location field-value. However, such an assertion cannot be trusted unless it can be verified by other means (not defined by this specification).
5. Otherwise, the payload is unidentified.

3.1.4.2. Content-Location

The "Content-Location" header field references a URI that can be used as an identifier for a specific resource corresponding to the representation in this message's payload. In other words, if one were to perform a GET request on this URI at the time of this message's generation, then a 200 (OK) response would contain the same representation that is enclosed as payload in this message.

Content-Location = absolute-URI / partial-URI

The Content-Location value is not a replacement for the effective Request URI (Section 5.5 of [Part1]). It is representation metadata. It has the same syntax and semantics as the header field of the same name defined for MIME body parts in Section 4 of [RFC2557]. However, its appearance in an HTTP message has some special implications for HTTP recipients.

If Content-Location is included in a 2xx (Successful) response message and its value refers (after conversion to absolute form) to a URI that is the same as the effective request URI, then the recipient MAY consider the payload to be a current representation of that resource at the time indicated by the message origination date. For a GET (Section 4.3.1) or HEAD (Section 4.3.2) request, this is the same as the default semantics when no Content-Location is provided by the server. For a state-changing request like PUT (Section 4.3.4) or POST (Section 4.3.3), it implies that the server's response contains the new representation of that resource, thereby distinguishing it

from representations that might only report about the action (e.g., "It worked!"). This allows authoring applications to update their local copies without the need for a subsequent GET request.

If Content-Location is included in a 2xx (Successful) response message and its field-value refers to a URI that differs from the effective request URI, then the origin server claims that the URI is an identifier for a different resource corresponding to the enclosed representation. Such a claim can only be trusted if both identifiers share the same resource owner, which cannot be programmatically determined via HTTP.

- o For a response to a GET or HEAD request, this is an indication that the effective request URI refers to a resource that is subject to content negotiation and the Content-Location field-value is a more specific identifier for the selected representation.
- o For a 201 (Created) response to a state-changing method, a Content-Location field-value that is identical to the Location field-value indicates that this payload is a current representation of the newly created resource.
- o Otherwise, such a Content-Location indicates that this payload is a representation reporting on the requested action's status and that the same report is available (for future access with GET) at the given URI. For example, a purchase transaction made via a POST request might include a receipt document as the payload of the 200 (OK) response; the Content-Location field-value provides an identifier for retrieving a copy of that same receipt in the future.

A user agent that sends Content-Location in a request message is stating that its value refers to where the user agent originally obtained the content of the enclosed representation (prior to any modifications made by that user agent). In other words, the user agent is providing a back link to the source of the original representation.

An origin server that receives a Content-Location field in a request message MUST treat the information as transitory request context rather than as metadata to be saved verbatim as part of the representation. An origin server MAY use that context to guide in processing the request or to save it for other uses, such as within source links or versioning metadata. However, an origin server MUST NOT use such context information to alter the request semantics.

For example, if a client makes a PUT request on a negotiated resource

and the origin server accepts that PUT (without redirection), then the new state of that resource is expected to be consistent with the one representation supplied in that PUT; the Content-Location cannot be used as a form of reverse content selection identifier to update only one of the negotiated representations. If the user agent had wanted the latter semantics, it would have applied the PUT directly to the Content-Location URI.

3.2. Representation Data

The representation data associated with an HTTP message is either provided as the payload body of the message or referred to by the message semantics and the effective request URI. The representation data is in a format and encoding defined by the representation metadata header fields.

The data type of the representation data is determined via the header fields Content-Type and Content-Encoding. These define a two-layer, ordered encoding model:

```
representation-data := Content-Encoding( Content-Type( bits ) )
```

3.3. Payload Semantics

Some HTTP messages transfer a complete or partial representation as the message "payload". In some cases, a payload might contain only the associated representation's header fields (e.g., responses to HEAD) or only some part(s) of the representation data (e.g., the 206 (Partial Content) status code).

The purpose of a payload in a request is defined by the method semantics. For example, a representation in the payload of a PUT request (Section 4.3.4) represents the desired state of the target resource if the request is successfully applied, whereas a representation in the payload of a POST request (Section 4.3.3) represents information to be processed by the target resource.

In a response, the payload's purpose is defined by both the request method and the response status code. For example, the payload of a 200 (OK) response to GET (Section 4.3.1) represents the current state of the target resource, as observed at the time of the message origination date (Section 7.1.1.2), whereas the payload of the same status code in a response to POST might represent either the processing result or the new state of the target resource after applying the processing. Response messages with an error status code usually contain a payload that represents the error condition, such that it describes the error state and what next steps are suggested for resolving it.

Header fields that specifically describe the payload, rather than the associated representation, are referred to as "payload header fields". Payload header fields are defined in other parts of this specification, due to their impact on message parsing.

| Header Field Name | Defined in... |
|-------------------|--------------------------|
| Content-Length | Section 3.3.2 of [Part1] |
| Content-Range | Section 4.2 of [Part5] |
| Trailer | Section 4.4 of [Part1] |
| Transfer-Encoding | Section 3.3.1 of [Part1] |

3.4. Content Negotiation

When responses convey payload information, whether indicating a success or an error, the origin server often has different ways of representing that information; for example, in different formats, languages, or encodings. Likewise, different users or user agents might have differing capabilities, characteristics, or preferences that could influence which representation, among those available, would be best to deliver. For this reason, HTTP provides mechanisms for content negotiation.

This specification defines two patterns of content negotiation that can be made visible within the protocol: "proactive", where the server selects the representation based upon the user agent's stated preferences, and "reactive" negotiation, where the server provides a list of representations for the user agent to choose from. Other patterns of content negotiation include "conditional content", where the representation consists of multiple parts that are selectively rendered based on user agent parameters, "active content", where the representation contains a script that makes additional (more specific) requests based on the user agent characteristics, and "Transparent Content Negotiation" ([RFC2295]), where content selection is performed by an intermediary. These patterns are not mutually exclusive, and each has trade-offs in applicability and practicality.

Note that, in all cases, HTTP is not aware of the resource semantics. The consistency with which an origin server responds to requests, over time and over the varying dimensions of content negotiation, and thus the "sameness" of a resource's observed representations over time, is determined entirely by whatever entity or algorithm selects or generates those responses. HTTP pays no attention to the man behind the curtain.

3.4.1. Proactive Negotiation

When content negotiation preferences are sent by the user agent in a request to encourage an algorithm located at the server to select the preferred representation, it is called proactive negotiation (a.k.a., server-driven negotiation). Selection is based on the available representations for a response (the dimensions over which it might vary, such as language, content-coding, etc.) compared to various information supplied in the request, including both the explicit negotiation fields of Section 5.3 and implicit characteristics, such as the client's network address or parts of the User-Agent field.

Proactive negotiation is advantageous when the algorithm for selecting from among the available representations is difficult to describe to a user agent, or when the server desires to send its "best guess" to the user agent along with the first response (hoping to avoid the round-trip delay of a subsequent request if the "best guess" is good enough for the user). In order to improve the server's guess, a user agent MAY send request header fields that describe its preferences.

Proactive negotiation has serious disadvantages:

- o It is impossible for the server to accurately determine what might be "best" for any given user, since that would require complete knowledge of both the capabilities of the user agent and the intended use for the response (e.g., does the user want to view it on screen or print it on paper?);
- o Having the user agent describe its capabilities in every request can be both very inefficient (given that only a small percentage of responses have multiple representations) and a potential risk to the user's privacy;
- o It complicates the implementation of an origin server and the algorithms for generating responses to a request; and,
- o It limits the reusability of responses for shared caching.

A user agent cannot rely on proactive negotiation preferences being consistently honored, since the origin server might not implement proactive negotiation for the requested resource or might decide that sending a response that doesn't conform to the user agent's preferences is better than sending a 406 (Not Acceptable) response.

A Vary header field (Section 7.1.4) is often sent in a response subject to proactive negotiation to indicate what parts of the request information were used in the selection algorithm.

3.4.2. Reactive Negotiation

With reactive negotiation (a.k.a., agent-driven negotiation), selection of the best response representation (regardless of the status code) is performed by the user agent after receiving an initial response from the origin server that contains a list of resources for alternative representations. If the user agent is not satisfied by the initial response representation, it can perform a GET request on one or more of the alternative resources, selected based on metadata included in the list, to obtain a different form of representation for that response. Selection of alternatives might be performed automatically by the user agent or manually by the user selecting from a generated (possibly hypertext) menu.

Note that the above refers to representations of the response, in general, not representations of the resource. The alternative representations are only considered representations of the target resource if the response in which those alternatives are provided has the semantics of being a representation of the target resource (e.g., a 200 (OK) response to a GET request) or has the semantics of providing links to alternative representations for the target resource (e.g., a 300 (Multiple Choices) response to a GET request).

A server might choose not to send an initial representation, other than the list of alternatives, and thereby indicate that reactive negotiation by the user agent is preferred. For example, the alternatives listed in responses with the 300 (Multiple Choices) and 406 (Not Acceptable) status codes include information about the available representations so that the user or user agent can react by making a selection.

Reactive negotiation is advantageous when the response would vary over commonly-used dimensions (such as type, language, or encoding), when the origin server is unable to determine a user agent's capabilities from examining the request, and generally when public caches are used to distribute server load and reduce network usage.

Reactive negotiation suffers from the disadvantages of transmitting a list of alternatives to the user agent, which degrades user-perceived latency if transmitted in the header section, and needing a second request to obtain an alternate representation. Furthermore, this specification does not define a mechanism for supporting automatic selection, though it does not prevent such a mechanism from being developed as an extension.

4. Request Methods

4.1. Overview

The request method token is the primary source of request semantics; it indicates the purpose for which the client has made this request and what is expected by the client as a successful result.

The request method's semantics might be further specialized by the semantics of some header fields when present in a request (Section 5) if those additional semantics do not conflict with the method. For example, a client can send conditional request header fields (Section 5.2) to make the requested action conditional on the current state of the target resource ([Part4]).

method = token

HTTP was originally designed to be usable as an interface to distributed object systems. The request method was envisioned as applying semantics to a target resource in much the same way as invoking a defined method on an identified object would apply semantics. The method token is case-sensitive because it might be used as a gateway to object-based systems with case-sensitive method names.

Unlike distributed objects, the standardized request methods in HTTP are not resource-specific, since uniform interfaces provide for better visibility and reuse in network-based systems [REST]. Once defined, a standardized method ought to have the same semantics when applied to any resource, though each resource determines for itself whether those semantics are implemented or allowed.

This specification defines a number of standardized methods that are commonly used in HTTP, as outlined by the following table. By convention, standardized methods are defined in all-uppercase ASCII letters.

| Method | Description | Sec. |
|---------|--|-------|
| GET | Transfer a current representation of the target resource. | 4.3.1 |
| HEAD | Same as GET, but only transfer the status line and header section. | 4.3.2 |
| POST | Perform resource-specific processing on the request payload. | 4.3.3 |
| PUT | Replace all current representations of the target resource with the request payload. | 4.3.4 |
| DELETE | Remove all current representations of the target resource. | 4.3.5 |
| CONNECT | Establish a tunnel to the server identified by the target resource. | 4.3.6 |
| OPTIONS | Describe the communication options for the target resource. | 4.3.7 |
| TRACE | Perform a message loop-back test along the path to the target resource. | 4.3.8 |

All general-purpose servers MUST support the methods GET and HEAD. All other methods are OPTIONAL.

Additional methods, outside the scope of this specification, have been standardized for use in HTTP. All such methods ought to be registered within the HTTP Method Registry maintained by IANA, as defined in Section 8.1.

The set of methods allowed by a target resource can be listed in an Allow header field (Section 7.4.1). However, the set of allowed methods can change dynamically. When a request method is received that is unrecognized or not implemented by an origin server, the origin server SHOULD respond with the 501 (Not Implemented) status code. When a request method is received that is known by an origin server but not allowed for the target resource, the origin server SHOULD respond with the 405 (Method Not Allowed) status code.

4.2. Common Method Properties

4.2.1. Safe Methods

Request methods are considered "safe" if their defined semantics are essentially read-only; i.e., the client does not request, and does not expect, any state change on the origin server as a result of applying a safe method to a target resource. Likewise, reasonable use of a safe method is not expected to cause any harm, loss of property, or unusual burden on the origin server.

This definition of safe methods does not prevent an implementation from including behavior that is potentially harmful, not entirely read-only, or which causes side-effects while invoking a safe method. What is important, however, is that the client did not request that additional behavior and cannot be held accountable for it. For example, most servers append request information to access log files at the completion of every response, regardless of the method, and that is considered safe even though the log storage might become full and crash the server. Likewise, a safe request initiated by selecting an advertisement on the Web will often have the side-effect of charging an advertising account.

Of the request methods defined by this specification, the GET, HEAD, OPTIONS, and TRACE methods are defined to be safe.

The purpose of distinguishing between safe and unsafe methods is to allow automated retrieval processes (spiders) and cache performance optimization (pre-fetching) to work without fear of causing harm. In addition, it allows a user agent to apply appropriate constraints on the automated use of unsafe methods when processing potentially untrusted content.

A user agent SHOULD distinguish between safe and unsafe methods when presenting potential actions to a user, such that the user can be made aware of an unsafe action before it is requested.

When a resource is constructed such that parameters within the effective request URI have the effect of selecting an action, it is the resource owner's responsibility to ensure that the action is consistent with the request method semantics. For example, it is common for Web-based content editing software to use actions within query parameters, such as "page?do=delete". If the purpose of such a resource is to perform an unsafe action, then the resource owner MUST disable or disallow that action when it is accessed using a safe request method. Failure to do so will result in unfortunate side-effects when automated processes perform a GET on every URI reference for the sake of link maintenance, pre-fetching, building a search index, etc.

4.2.2. Idempotent Methods

A request method is considered "idempotent" if the intended effect on the server of multiple identical requests with that method is the same as the effect for a single such request. Of the request methods defined by this specification, PUT, DELETE, and safe request methods are idempotent.

Like the definition of safe, the idempotent property only applies to

what has been requested by the user; a server is free to log each request separately, retain a revision control history, or implement other non-idempotent side-effects for each idempotent request.

Idempotent methods are distinguished because the request can be repeated automatically if a communication failure occurs before the client is able to read the server's response. For example, if a client sends a PUT request and the underlying connection is closed before any response is received, then the client can establish a new connection and retry the idempotent request. It knows that repeating the request will have the same intended effect, even if the original request succeeded, though the response might differ.

4.2.3. Cacheable Methods

Request methods can be defined as "cacheable" to indicate that responses to them are allowed to be stored for future reuse; for specific requirements see [Part6]. In general, safe methods that do not depend on a current or authoritative response are defined as cacheable; this specification defines GET, HEAD and POST as cacheable, although the overwhelming majority of cache implementations only support GET and HEAD.

4.3. Method Definitions

4.3.1. GET

The GET method requests transfer of a current selected representation for the target resource. GET is the primary mechanism of information retrieval and the focus of almost all performance optimizations. Hence, when people speak of retrieving some identifiable information via HTTP, they are generally referring to making a GET request.

It is tempting to think of resource identifiers as remote file system pathnames, and of representations as being a copy of the contents of such files. In fact, that is how many resources are implemented (see Section 9.1 for related security considerations). However, there are no such limitations in practice. The HTTP interface for a resource is just as likely to be implemented as a tree of content objects, a programmatic view on various database records, or a gateway to other information systems. Even when the URI mapping mechanism is tied to a file system, an origin server might be configured to execute the files with the request as input and send the output as the representation, rather than transfer the files directly. Regardless, only the origin server needs to know how each of its resource identifiers corresponds to an implementation, and how each implementation manages to select and send a current representation of the target resource in a response to GET.

A client can alter the semantics of GET to be a "range request", requesting transfer of only some part(s) of the selected representation, by sending a Range header field in the request ([Part5]).

A payload within a GET request message has no defined semantics; sending a payload body on a GET request might cause some existing implementations to reject the request.

The response to a GET request is cacheable; a cache MAY use it to satisfy subsequent GET and HEAD requests unless otherwise indicated by the Cache-Control header field (Section 5.2 of [Part6]).

4.3.2. HEAD

The HEAD method is identical to GET except that the server MUST NOT send a message body in the response (i.e., the response terminates at the end of the header section). The server SHOULD send the same header fields in response to a HEAD request as it would have sent if the request had been a GET, except that the payload header fields (Section 3.3) MAY be omitted. This method can be used for obtaining metadata about the selected representation without transferring the representation data and is often used for testing hypertext links for validity, accessibility, and recent modification.

A payload within a HEAD request message has no defined semantics; sending a payload body on a HEAD request might cause some existing implementations to reject the request.

The response to a HEAD request is cacheable; a cache MAY use it to satisfy subsequent HEAD requests unless otherwise indicated by the Cache-Control header field (Section 5.2 of [Part6]). A HEAD response might also have an effect on previously cached responses to GET; see Section 4.3.5 of [Part6].

4.3.3. POST

The POST method requests that the target resource process the representation enclosed in the request according to the resource's own specific semantics. For example, POST is used for the following functions (among others):

- o Providing a block of data, such as the fields entered into an HTML form, to a data-handling process;
- o Posting a message to a bulletin board, newsgroup, mailing list, blog, or similar group of articles;

- o Creating a new resource that has yet to be identified by the origin server; and
- o Appending data to a resource's existing representation(s).

An origin server indicates response semantics by choosing an appropriate status code depending on the result of processing the POST request; almost all of the status codes defined by this specification might be received in a response to POST (the exceptions being 206, 304, and 416).

If one or more resources has been created on the origin server as a result of successfully processing a POST request, the origin server SHOULD send a 201 (Created) response containing a Location header field that provides an identifier for the primary resource created (Section 7.1.2) and a representation that describes the status of the request while referring to the new resource(s).

Responses to POST requests are only cacheable when they include explicit freshness information (see Section 4.2.1 of [Part6]). However, POST caching is not widely implemented. For cases where an origin server wishes the client to be able to cache the result of a POST in a way that can be reused by a later GET, the origin server MAY send a 200 (OK) response containing the result and a Content-Location header field that has the same value as the POST's effective request URI (Section 3.1.4.2).

If the result of processing a POST would be equivalent to a representation of an existing resource, an origin server MAY redirect the user agent to that resource by sending a 303 (See Other) response with the existing resource's identifier in the Location field. This has the benefits of providing the user agent a resource identifier and transferring the representation via a method more amenable to shared caching, though at the cost of an extra request if the user agent does not already have the representation cached.

4.3.4. PUT

The PUT method requests that the state of the target resource be created or replaced with the state defined by the representation enclosed in the request message payload. A successful PUT of a given representation would suggest that a subsequent GET on that same target resource will result in an equivalent representation being sent in a 200 (OK) response. However, there is no guarantee that such a state change will be observable, since the target resource might be acted upon by other user agents in parallel, or might be subject to dynamic processing by the origin server, before any subsequent GET is received. A successful response only implies that

the user agent's intent was achieved at the time of its processing by the origin server.

If the target resource does not have a current representation and the PUT successfully creates one, then the origin server MUST inform the user agent by sending a 201 (Created) response. If the target resource does have a current representation and that representation is successfully modified in accordance with the state of the enclosed representation, then the origin server MUST send either a 200 (OK) or a 204 (No Content) response to indicate successful completion of the request.

An origin server SHOULD ignore unrecognized header fields received in a PUT request (i.e., do not save them as part of the resource state).

An origin server SHOULD verify that the PUT representation is consistent with any constraints the server has for the target resource that cannot or will not be changed by the PUT. This is particularly important when the origin server uses internal configuration information related to the URI in order to set the values for representation metadata on GET responses. When a PUT representation is inconsistent with the target resource, the origin server SHOULD either make them consistent, by transforming the representation or changing the resource configuration, or respond with an appropriate error message containing sufficient information to explain why the representation is unsuitable. The 409 (Conflict) or 415 (Unsupported Media Type) status codes are suggested, with the latter being specific to constraints on Content-Type values.

For example, if the target resource is configured to always have a Content-Type of "text/html" and the representation being PUT has a Content-Type of "image/jpeg", the origin server ought to do one of:

- a. reconfigure the target resource to reflect the new media type;
- b. transform the PUT representation to a format consistent with that of the resource before saving it as the new resource state; or,
- c. reject the request with a 415 (Unsupported Media Type) response indicating that the target resource is limited to "text/html", perhaps including a link to a different resource that would be a suitable target for the new representation.

HTTP does not define exactly how a PUT method affects the state of an origin server beyond what can be expressed by the intent of the user agent request and the semantics of the origin server response. It does not define what a resource might be, in any sense of that word, beyond the interface provided via HTTP. It does not define how

resource state is "stored", nor how such storage might change as a result of a change in resource state, nor how the origin server translates resource state into representations. Generally speaking, all implementation details behind the resource interface are intentionally hidden by the server.

An origin server **MUST NOT** send a validator header field (Section 7.2), such as an ETag or Last-Modified field, in a successful response to PUT unless the request's representation data was saved without any transformation applied to the body (i.e., the resource's new representation data is identical to the representation data received in the PUT request) and the validator field value reflects the new representation. This requirement allows a user agent to know when the representation body it has in memory remains current as a result of the PUT, thus not in need of retrieving again from the origin server, and that the new validator(s) received in the response can be used for future conditional requests in order to prevent accidental overwrites (Section 5.2).

The fundamental difference between the POST and PUT methods is highlighted by the different intent for the enclosed representation. The target resource in a POST request is intended to handle the enclosed representation according to the resource's own semantics, whereas the enclosed representation in a PUT request is defined as replacing the state of the target resource. Hence, the intent of PUT is idempotent and visible to intermediaries, even though the exact effect is only known by the origin server.

Proper interpretation of a PUT request presumes that the user agent knows which target resource is desired. A service that selects a proper URI on behalf of the client, after receiving a state-changing request, **SHOULD** be implemented using the POST method rather than PUT. If the origin server will not make the requested PUT state change to the target resource and instead wishes to have it applied to a different resource, such as when the resource has been moved to a different URI, then the origin server **MUST** send an appropriate 3xx (Redirection) response; the user agent **MAY** then make its own decision regarding whether or not to redirect the request.

A PUT request applied to the target resource can have side-effects on other resources. For example, an article might have a URI for identifying "the current version" (a resource) that is separate from the URIs identifying each particular version (different resources that at one point shared the same state as the current version resource). A successful PUT request on "the current version" URI might therefore create a new version resource in addition to changing the state of the target resource, and might also cause links to be added between the related resources.

An origin server that allows PUT on a given target resource **MUST** send a 400 (Bad Request) response to a PUT request that contains a Content-Range header field (Section 4.2 of [Part5]), since the payload is likely to be partial content that has been mistakenly PUT as a full representation. Partial content updates are possible by targeting a separately identified resource with state that overlaps a portion of the larger resource, or by using a different method that has been specifically defined for partial updates (for example, the PATCH method defined in [RFC5789]).

Responses to the PUT method are not cacheable. If a successful PUT request passes through a cache that has one or more stored responses for the effective request URI, those stored responses will be invalidated (see Section 4.4 of [Part6]).

4.3.5. DELETE

The DELETE method requests that the origin server remove the association between the target resource and its current functionality. In effect, this method is similar to the `rm` command in UNIX: it expresses a deletion operation on the URI mapping of the origin server, rather than an expectation that the previously associated information be deleted.

If the target resource has one or more current representations, they might or might not be destroyed by the origin server, and the associated storage might or might not be reclaimed, depending entirely on the nature of the resource and its implementation by the origin server (which are beyond the scope of this specification). Likewise, other implementation aspects of a resource might need to be deactivated or archived as a result of a DELETE, such as database or gateway connections. In general, it is assumed that the origin server will only allow DELETE on resources for which it has a prescribed mechanism for accomplishing the deletion.

Relatively few resources allow the DELETE method -- its primary use is for remote authoring environments, where the user has some direction regarding its effect. For example, a resource that was previously created using a PUT request, or identified via the Location header field after a 201 (Created) response to a POST request, might allow a corresponding DELETE request to undo those actions. Similarly, custom user agent implementations that implement an authoring function, such as revision control clients using HTTP for remote operations, might use DELETE based on an assumption that the server's URI space has been crafted to correspond to a version repository.

If a DELETE method is successfully applied, the origin server **SHOULD**

send a 202 (Accepted) status code if the action will likely succeed but has not yet been enacted, a 204 (No Content) status code if the action has been enacted and no further information is to be supplied, or a 200 (OK) status code if the action has been enacted and the response message includes a representation describing the status.

A payload within a DELETE request message has no defined semantics; sending a payload body on a DELETE request might cause some existing implementations to reject the request.

Responses to the DELETE method are not cacheable. If a DELETE request passes through a cache that has one or more stored responses for the effective request URI, those stored responses will be invalidated (see Section 4.4 of [Part6]).

4.3.6. CONNECT

The CONNECT method requests that the recipient establish a tunnel to the destination origin server identified by the request-target and, if successful, thereafter restrict its behavior to blind forwarding of packets, in both directions, until the tunnel is closed. Tunnels are commonly used to create an end-to-end virtual connection, through one or more proxies, which can then be secured using TLS (Transport Layer Security, [RFC5246]).

CONNECT is intended only for use in requests to a proxy. An origin server that receives a CONNECT request for itself MAY respond with a 2xx status code to indicate that a connection is established. However, most origin servers do not implement CONNECT.

A client sending a CONNECT request MUST send the authority form of request-target (Section 5.3 of [Part1]); i.e., the request-target consists of only the host name and port number of the tunnel destination, separated by a colon. For example,

```
CONNECT server.example.com:80 HTTP/1.1
Host: server.example.com:80
```

The recipient proxy can establish a tunnel either by directly connecting to the request-target or, if configured to use another proxy, by forwarding the CONNECT request to the next inbound proxy. Any 2xx (Successful) response indicates that the sender (and all inbound proxies) will switch to tunnel mode immediately after the blank line that concludes the successful response's header section; data received after that blank line is from the server identified by the request-target. Any response other than a successful response indicates that the tunnel has not yet been formed and that the

connection remains governed by HTTP.

A tunnel is closed when a tunnel intermediary detects that either side has closed its connection: the intermediary **MUST** attempt to send any outstanding data that came from the closed side to the other side, close both connections, and then discard any remaining data left undelivered.

Proxy authentication might be used to establish the authority to create a tunnel. For example,

```
CONNECT server.example.com:80 HTTP/1.1
Host: server.example.com:80
Proxy-Authorization: basic aGVsbG86d29ybGQ=
```

There are significant risks in establishing a tunnel to arbitrary servers, particularly when the destination is a well-known or reserved TCP port that is not intended for Web traffic. For example, a **CONNECT** to a request-target of "example.com:25" would suggest that the proxy connect to the reserved port for SMTP traffic; if allowed, that could trick the proxy into relaying spam email. Proxies that support **CONNECT** **SHOULD** restrict its use to a limited set of known ports or a configurable whitelist of safe request targets.

A server **MUST NOT** send any Transfer-Encoding or Content-Length header fields in a 2xx (Successful) response to **CONNECT**. A client **MUST** ignore any Content-Length or Transfer-Encoding header fields received in a successful response to **CONNECT**.

A payload within a **CONNECT** request message has no defined semantics; sending a payload body on a **CONNECT** request might cause some existing implementations to reject the request.

Responses to the **CONNECT** method are not cacheable.

4.3.7. OPTIONS

The **OPTIONS** method requests information about the communication options available for the target resource, either at the origin server or an intervening intermediary. This method allows a client to determine the options and/or requirements associated with a resource, or the capabilities of a server, without implying a resource action.

An **OPTIONS** request with an asterisk ("*") as the request-target (Section 5.3 of [Part1]) applies to the server in general rather than to a specific resource. Since a server's communication options

typically depend on the resource, the "*" request is only useful as a "ping" or "no-op" type of method; it does nothing beyond allowing the client to test the capabilities of the server. For example, this can be used to test a proxy for HTTP/1.1 conformance (or lack thereof).

If the request-target is not an asterisk, the OPTIONS request applies to the options that are available when communicating with the target resource.

A server generating a successful response to OPTIONS SHOULD send any header fields that might indicate optional features implemented by the server and applicable to the target resource (e.g., Allow), including potential extensions not defined by this specification. The response payload, if any, might also describe the communication options in a machine or human-readable representation. A standard format for such a representation is not defined by this specification, but might be defined by future extensions to HTTP. A server MUST generate a Content-Length field with a value of "0" if no payload body is to be sent in the response.

A client MAY send a Max-Forwards header field in an OPTIONS request to target a specific recipient in the request chain (see Section 5.1.2). A proxy MUST NOT generate a Max-Forwards header field while forwarding a request unless that request was received with a Max-Forwards field.

A client that generates an OPTIONS request containing a payload body MUST send a valid Content-Type header field describing the representation media type. Although this specification does not define any use for such a payload, future extensions to HTTP might use the OPTIONS body to make more detailed queries about the target resource.

Responses to the OPTIONS method are not cacheable.

4.3.8. TRACE

The TRACE method requests a remote, application-level loop-back of the request message. The final recipient of the request SHOULD reflect the message received, excluding some fields described below, back to the client as the message body of a 200 (OK) response with a Content-Type of "message/http" (Section 8.3.1 of [Part1]). The final recipient is either the origin server or the first server to receive a Max-Forwards value of zero (0) in the request (Section 5.1.2).

A client MUST NOT generate header fields in a TRACE request containing sensitive data that might be disclosed by the response. For example, it would be foolish for a user agent to send stored user

credentials [Part7] or cookies [RFC6265] in a TRACE request. The final recipient of the request SHOULD exclude any request header fields that are likely to contain sensitive data when that recipient generates the response body.

TRACE allows the client to see what is being received at the other end of the request chain and use that data for testing or diagnostic information. The value of the Via header field (Section 5.7.1 of [Part1]) is of particular interest, since it acts as a trace of the request chain. Use of the Max-Forwards header field allows the client to limit the length of the request chain, which is useful for testing a chain of proxies forwarding messages in an infinite loop.

A client MUST NOT send a message body in a TRACE request.

Responses to the TRACE method are not cacheable.

5. Request Header Fields

A client sends request header fields to provide more information about the request context, make the request conditional based on the target resource state, suggest preferred formats for the response, supply authentication credentials, or modify the expected request processing. These fields act as request modifiers, similar to the parameters on a programming language method invocation.

5.1. Controls

Controls are request header fields that direct specific handling of the request.

| Header Field Name | Defined in... |
|-------------------|------------------------|
| Cache-Control | Section 5.2 of [Part6] |
| Expect | Section 5.1.1 |
| Host | Section 5.4 of [Part1] |
| Max-Forwards | Section 5.1.2 |
| Pragma | Section 5.4 of [Part6] |
| Range | Section 3.1 of [Part5] |
| TE | Section 4.3 of [Part1] |

5.1.1. Expect

The "Expect" header field in a request indicates a certain set of behaviors (expectations) that need to be supported by the server in order to properly handle this request. The only such expectation

defined by this specification is 100-continue.

```
Expect = "100-continue"
```

The Expect field-value is case-insensitive.

A server that receives an Expect field-value other than 100-continue MAY respond with a 417 (Expectation Failed) status code to indicate that the unexpected expectation cannot be met.

A 100-continue expectation informs recipients that the client is about to send a (presumably large) message body in this request and wishes to receive a 100 (Continue) interim response if the request-line and header fields are not sufficient to cause an immediate success, redirect, or error response. This allows the client to wait for an indication that it is worthwhile to send the message body before actually doing so, which can improve efficiency when the message body is huge or when the client anticipates that an error is likely (e.g., when sending a state-changing method, for the first time, without previously verified authentication credentials).

For example, a request that begins with

```
PUT /somewhere/fun HTTP/1.1
Host: origin.example.com
Content-Type: video/h264
Content-Length: 1234567890987
Expect: 100-continue
```

allows the origin server to immediately respond with an error message, such as 401 (Unauthorized) or 405 (Method Not Allowed), before the client starts filling the pipes with an unnecessary data transfer.

Requirements for clients:

- o A client MUST NOT generate a 100-continue expectation in a request that does not include a message body.
- o A client that will wait for a 100 (Continue) response before sending the request message body MUST send an Expect header field containing a 100-continue expectation.
- o A client that sends a 100-continue expectation is not required to wait for any specific length of time; such a client MAY proceed to send the message body even if it has not yet received a response. Furthermore, since 100 (Continue) responses cannot be sent through

an HTTP/1.0 intermediary, such a client SHOULD NOT wait for an indefinite period before sending the message body.

- o A client that receives a 417 (Expectation Failed) status code in response to a request containing a 100-continue expectation SHOULD repeat that request without a 100-continue expectation, since the 417 response merely indicates that the response chain does not support expectations (e.g., it passes through an HTTP/1.0 server).

Requirements for servers:

- o A server that receives a 100-continue expectation in an HTTP/1.0 request MUST ignore that expectation.
- o A server MAY omit sending a 100 (Continue) response if it has already received some or all of the message body for the corresponding request, or if the framing indicates that there is no message body.
- o A server that sends a 100 (Continue) response MUST ultimately send a final status code, once the message body is received and processed, unless the connection is closed prematurely.
- o A server that responds with a final status code before reading the entire message body SHOULD indicate in that response whether it intends to close the connection or continue reading and discarding the request message (see Section 6.6 of [Part1]).

An origin server MUST, upon receiving an HTTP/1.1 (or later) request-line and a complete header section that contains a 100-continue expectation and indicates a request message body will follow, either send an immediate response with a final status code, if that status can be determined by examining just the request-line and header fields, or send an immediate 100 (Continue) response to encourage the client to send the request's message body. The origin server MUST NOT wait for the message body before sending the 100 (Continue) response.

A proxy MUST, upon receiving an HTTP/1.1 (or later) request-line and a complete header section that contains a 100-continue expectation and indicates a request message body will follow, either send an immediate response with a final status code, if that status can be determined by examining just the request-line and header fields, or begin forwarding the request toward the origin server by sending a corresponding request-line and header section to the next inbound server. If the proxy believes (from configuration or past interaction) that the next inbound server only supports HTTP/1.0, the proxy MAY generate an immediate 100 (Continue) response to encourage

the client to begin sending the message body.

Note: The Expect header field was added after the original publication of HTTP/1.1 [RFC2068] as both the means to request an interim 100 response and the general mechanism for indicating must-understand extensions. However, the extension mechanism has not been used by clients and the must-understand requirements have not been implemented by many servers, rendering the extension mechanism useless. This specification has removed the extension mechanism in order to simplify the definition and processing of 100-continue.

5.1.2. Max-Forwards

The "Max-Forwards" header field provides a mechanism with the TRACE (Section 4.3.8) and OPTIONS (Section 4.3.7) request methods to limit the number of times that the request is forwarded by proxies. This can be useful when the client is attempting to trace a request that appears to be failing or looping mid-chain.

Max-Forwards = 1*DIGIT

The Max-Forwards value is a decimal integer indicating the remaining number of times this request message can be forwarded.

Each intermediary that receives a TRACE or OPTIONS request containing a Max-Forwards header field MUST check and update its value prior to forwarding the request. If the received value is zero (0), the intermediary MUST NOT forward the request; instead, the intermediary MUST respond as the final recipient. If the received Max-Forwards value is greater than zero, the intermediary MUST generate an updated Max-Forwards field in the forwarded message with a field-value that is the lesser of: a) the received value decremented by one (1), or b) the recipient's maximum supported value for Max-Forwards.

A recipient MAY ignore a Max-Forwards header field received with any other request methods.

5.2. Conditionals

The HTTP conditional request header fields [Part4] allow a client to place a precondition on the state of the target resource, so that the action corresponding to the method semantics will not be applied if the precondition evaluates to false. Each precondition defined by this specification consists of a comparison between a set of validators obtained from prior representations of the target resource to the current state of validators for the selected representation (Section 7.2). Hence, these preconditions evaluate whether the state

of the target resource has changed since a given state known by the client. The effect of such an evaluation depends on the method semantics and choice of conditional, as defined in Section 5 of [Part4].

| Header Field Name | Defined in... |
|---------------------|------------------------|
| If-Match | Section 3.1 of [Part4] |
| If-None-Match | Section 3.2 of [Part4] |
| If-Modified-Since | Section 3.3 of [Part4] |
| If-Unmodified-Since | Section 3.4 of [Part4] |
| If-Range | Section 3.2 of [Part5] |

5.3. Content Negotiation

The following request header fields are sent by a user agent to engage in proactive negotiation of the response content, as defined in Section 3.4.1. The preferences sent in these fields apply to any content in the response, including representations of the target resource, representations of error or processing status, and potentially even the miscellaneous text strings that might appear within the protocol.

| Header Field Name | Defined in... |
|-------------------|---------------|
| Accept | Section 5.3.2 |
| Accept-Charset | Section 5.3.3 |
| Accept-Encoding | Section 5.3.4 |
| Accept-Language | Section 5.3.5 |

5.3.1. Quality Values

Many of the request header fields for proactive negotiation use a common parameter, named "q" (case-insensitive), to assign a relative "weight" to the preference for that associated kind of content. This weight is referred to as a "quality value" (or "qvalue") because the same parameter name is often used within server configurations to assign a weight to the relative quality of the various representations that can be selected for a resource.

The weight is normalized to a real number in the range 0 through 1, where 0.001 is the least preferred and 1 is the most preferred; a value of 0 means "not acceptable". If no "q" parameter is present, the default weight is 1.

```

weight = OWS ";" OWS "q=" qvalue
qvalue = ( "0" [ "." 0*3DIGIT ] )
        / ( "1" [ "." 0*3("0") ] )

```

A sender of qvalue MUST NOT generate more than three digits after the decimal point. User configuration of these values ought to be limited in the same fashion.

5.3.2. Accept

The "Accept" header field can be used by user agents to specify response media types that are acceptable. Accept header fields can be used to indicate that the request is specifically limited to a small set of desired types, as in the case of a request for an in-line image.

```

Accept = #( media-range [ accept-params ] )

media-range    = ( "*"/*"
                  / ( type "/" "*" )
                  / ( type "/" subtype )
                  ) *( OWS ";" OWS parameter )
accept-params  = weight *( accept-ext )
accept-ext     = OWS ";" OWS token [ "=" ( token / quoted-string ) ]

```

The asterisk "*" character is used to group media types into ranges, with "*"/*" indicating all media types and "type/*" indicating all subtypes of that type. The media-range can include media type parameters that are applicable to that range.

Each media-range might be followed by zero or more applicable media type parameters (e.g., charset), an optional "q" parameter for indicating a relative weight (Section 5.3.1), and then zero or more extension parameters. The "q" parameter is necessary if any extensions (accept-ext) are present, since it acts as a separator between the two parameter sets.

Note: Use of the "q" parameter name to separate media type parameters from Accept extension parameters is due to historical practice. Although this prevents any media type parameter named "q" from being used with a media range, such an event is believed to be unlikely given the lack of any "q" parameters in the IANA media type registry and the rare usage of any media type parameters in Accept. Future media types are discouraged from registering any parameter named "q".

The example

```
Accept: audio/*; q=0.2, audio/basic
```

is interpreted as "I prefer audio/basic, but send me any audio type if it is the best available after an 80% mark-down in quality".

A request without any Accept header field implies that the user agent will accept any media type in response. If the header field is present in a request and none of the available representations for the response have a media type that is listed as acceptable, the origin server can either honor the header field by sending a 406 (Not Acceptable) response or disregard the header field by treating the response as if it is not subject to content negotiation.

A more elaborate example is

```
Accept: text/plain; q=0.5, text/html,  
       text/x-dvi; q=0.8, text/x-c
```

Verbally, this would be interpreted as "text/html and text/x-c are the equally preferred media types, but if they do not exist, then send the text/x-dvi representation, and if that does not exist, send the text/plain representation".

Media ranges can be overridden by more specific media ranges or specific media types. If more than one media range applies to a given type, the most specific reference has precedence. For example,

```
Accept: text/*, text/plain, text/plain;format=flowed, */*
```

have the following precedence:

1. text/plain;format=flowed
2. text/plain
3. text/*
4. */*

The media type quality factor associated with a given type is determined by finding the media range with the highest precedence that matches the type. For example,

```
Accept: text/*;q=0.3, text/html;q=0.7, text/html;level=1,  
       text/html;level=2;q=0.4, */*;q=0.5
```

would cause the following values to be associated:

| Media Type | Quality Value |
|-------------------|---------------|
| text/html;level=1 | 1 |
| text/html | 0.7 |
| text/plain | 0.3 |
| image/jpeg | 0.5 |
| text/html;level=2 | 0.4 |
| text/html;level=3 | 0.7 |

Note: A user agent might be provided with a default set of quality values for certain media ranges. However, unless the user agent is a closed system that cannot interact with other rendering agents, this default set ought to be configurable by the user.

5.3.3. Accept-Charset

The "Accept-Charset" header field can be sent by a user agent to indicate what charsets are acceptable in textual response content. This field allows user agents capable of understanding more comprehensive or special-purpose charsets to signal that capability to an origin server that is capable of representing information in those charsets.

Accept-Charset = 1#((charset / "*") [weight])

Charset names are defined in Section 3.1.1.2. A user agent MAY associate a quality value with each charset to indicate the user's relative preference for that charset, as defined in Section 5.3.1. An example is

Accept-Charset: iso-8859-5, unicode-1-1;q=0.8

The special value "*", if present in the Accept-Charset field, matches every charset that is not mentioned elsewhere in the Accept-Charset field. If no "*" is present in an Accept-Charset field, then any charsets not explicitly mentioned in the field are considered "not acceptable" to the client.

A request without any Accept-Charset header field implies that the user agent will accept any charset in response. Most general-purpose user agents do not send Accept-Charset, unless specifically configured to do so, because a detailed list of supported charsets makes it easier for a server to identify an individual by virtue of the user agent's request characteristics (Section 9.7).

If an Accept-Charset header field is present in a request and none of

the available representations for the response has a charset that is listed as acceptable, the origin server can either honor the header field, by sending a 406 (Not Acceptable) response, or disregard the header field by treating the resource as if it is not subject to content negotiation.

5.3.4. Accept-Encoding

The "Accept-Encoding" header field can be used by user agents to indicate what response content-codings (Section 3.1.2.1) are acceptable in the response. An "identity" token is used as a synonym for "no encoding" in order to communicate when no encoding is preferred.

```
Accept-Encoding = #( codings [ weight ] )
codings         = content-coding / "identity" / "*"
```

Each codings value MAY be given an associated quality value representing the preference for that encoding, as defined in Section 5.3.1. The asterisk "*" symbol in an Accept-Encoding field matches any available content-coding not explicitly listed in the header field.

For example,

```
Accept-Encoding: compress, gzip
Accept-Encoding:
Accept-Encoding: *
Accept-Encoding: compress;q=0.5, gzip;q=1.0
Accept-Encoding: gzip;q=1.0, identity; q=0.5, *;q=0
```

A request without an Accept-Encoding header field implies that the user agent has no preferences regarding content-codings. Although this allows the server to use any content-coding in a response, it does not imply that the user agent will be able to correctly process all encodings.

A server tests whether a content-coding for a given representation is acceptable using these rules:

1. If no Accept-Encoding field is in the request, any content-coding is considered acceptable by the user agent.
2. If the representation has no content-coding, then it is acceptable by default unless specifically excluded by the Accept-Encoding field stating either "identity;q=0" or "*;q=0" without a more specific entry for "identity".

3. If the representation's content-coding is one of the content-codings listed in the Accept-Encoding field, then it is acceptable unless it is accompanied by a qvalue of 0. (As defined in Section 5.3.1, a qvalue of 0 means "not acceptable".)
4. If multiple content-codings are acceptable, then the acceptable content-coding with the highest non-zero qvalue is preferred.

An Accept-Encoding header field with a combined field-value that is empty implies that the user agent does not want any content-coding in response. If an Accept-Encoding header field is present in a request and none of the available representations for the response have a content-coding that is listed as acceptable, the origin server SHOULD send a response without any content-coding.

Note: Most HTTP/1.0 applications do not recognize or obey qvalues associated with content-codings. This means that qvalues might not work and are not permitted with x-gzip or x-compress.

5.3.5. Accept-Language

The "Accept-Language" header field can be used by user agents to indicate the set of natural languages that are preferred in the response. Language tags are defined in Section 3.1.3.1.

```
Accept-Language = 1#( language-range [ weight ] )  
language-range =  
    <language-range, defined in [RFC4647], Section 2.1>
```

Each language-range can be given an associated quality value representing an estimate of the user's preference for the languages specified by that range, as defined in Section 5.3.1. For example,

```
Accept-Language: da, en-gb;q=0.8, en;q=0.7
```

would mean: "I prefer Danish, but will accept British English and other types of English".

A request without any Accept-Language header field implies that the user agent will accept any language in response. If the header field is present in a request and none of the available representations for the response have a matching language tag, the origin server can either disregard the header field by treating the response as if it is not subject to content negotiation, or honor the header field by sending a 406 (Not Acceptable) response. However, the latter is not encouraged, as doing so can prevent users from accessing content that they might be able to use (with translation software, for example).

Note that some recipients treat the order in which language tags are listed as an indication of descending priority, particularly for tags that are assigned equal quality values (no value is the same as $q=1$). However, this behavior cannot be relied upon. For consistency and to maximize interoperability, many user agents assign each language tag a unique quality value while also listing them in order of decreasing quality. Additional discussion of language priority lists can be found in Section 2.3 of [RFC4647].

For matching, Section 3 of [RFC4647] defines several matching schemes. Implementations can offer the most appropriate matching scheme for their requirements. The "Basic Filtering" scheme ([RFC4647], Section 3.3.1) is identical to the matching scheme that was previously defined for HTTP in Section 14.4 of [RFC2616].

It might be contrary to the privacy expectations of the user to send an Accept-Language header field with the complete linguistic preferences of the user in every request (Section 9.7).

Since intelligibility is highly dependent on the individual user, user agents need to allow user control over the linguistic preference (either through configuration of the user agent itself, or by defaulting to a user controllable system setting). A user agent that does not provide such control to the user **MUST NOT** send an Accept-Language header field.

Note: User agents ought to provide guidance to users when setting a preference, since users are rarely familiar with the details of language matching as described above. For example, users might assume that on selecting "en-gb", they will be served any kind of English document if British English is not available. A user agent might suggest, in such a case, to add "en" to the list for better matching behavior.

5.4. Authentication Credentials

Two header fields are used for carrying authentication credentials, as defined in [Part7]. Note that various custom mechanisms for user authentication use the Cookie header field for this purpose, as defined in [RFC6265].

| Header Field Name | Defined in... |
|---------------------|------------------------|
| Authorization | Section 4.2 of [Part7] |
| Proxy-Authorization | Section 4.4 of [Part7] |

5.5. Request Context

The following request header fields provide additional information about the request context, including information about the user, user agent, and resource behind the request.

| Header Field Name | Defined in... |
|-------------------|---------------|
| From | Section 5.5.1 |
| Referer | Section 5.5.2 |
| User-Agent | Section 5.5.3 |

5.5.1. From

The "From" header field contains an Internet email address for a human user who controls the requesting user agent. The address ought to be machine-usable, as defined by "mailbox" in Section 3.4 of [RFC5322]:

From = mailbox

mailbox = <mailbox, defined in [RFC5322], Section 3.4>

An example is:

From: webmaster@example.org

The From header field is rarely sent by non-robotic user agents. A user agent SHOULD NOT send a From header field without explicit configuration by the user, since that might conflict with the user's privacy interests or their site's security policy.

A robotic user agent SHOULD send a valid From header field so that the person responsible for running the robot can be contacted if problems occur on servers, such as if the robot is sending excessive, unwanted, or invalid requests.

A server SHOULD NOT use the From header field for access control or authentication, since most recipients will assume that the field value is public information.

5.5.2. Referer

The "Referer" [sic] header field allows the user agent to specify a URI reference for the resource from which the target URI was obtained (i.e., the "referrer", though the field name is misspelled). A user

agent **MUST NOT** include the fragment and userinfo components of the URI reference [RFC3986], if any, when generating the Referer field value.

Referer = absolute-URI / partial-URI

The Referer header field allows servers to generate back-links to other resources for simple analytics, logging, optimized caching, etc. It also allows obsolete or mistyped links to be found for maintenance. Some servers use the Referer header field as a means of denying links from other sites (so-called "deep linking") or restricting cross-site request forgery (CSRF), but not all requests contain it.

Example:

Referer: <http://www.example.org/hypertext/Overview.html>

If the target URI was obtained from a source that does not have its own URI (e.g., input from the user keyboard, or an entry within the user's bookmarks/favorites), the user agent **MUST** either exclude Referer or send it with a value of "about:blank".

The Referer field has the potential to reveal information about the request context or browsing history of the user, which is a privacy concern if the referring resource's identifier reveals personal information (such as an account name) or a resource that is supposed to be confidential (such as behind a firewall or internal to a secured service). Most general-purpose user agents do not send the Referer header field when the referring resource is a local "file" or "data" URI. A user agent **MUST NOT** send a Referer header field in an unsecured HTTP request if the referring page was received with a secure protocol. See Section 9.4 for additional security considerations.

Some intermediaries have been known to indiscriminately remove Referer header fields from outgoing requests. This has the unfortunate side-effect of interfering with protection against CSRF attacks, which can be far more harmful to their users. Intermediaries and user agent extensions that wish to limit information disclosure in Referer ought to restrict their changes to specific edits, such as replacing internal domain names with pseudonyms or truncating the query and/or path components. An intermediary **SHOULD NOT** modify or delete the Referer header field when the field value shares the same scheme and host as the request target.

5.5.3. User-Agent

The "User-Agent" header field contains information about the user agent originating the request, which is often used by servers to help identify the scope of reported interoperability problems, to work around or tailor responses to avoid particular user agent limitations, and for analytics regarding browser or operating system use. A user agent **SHOULD** send a User-Agent field in each request unless specifically configured not to do so.

User-Agent = product *(RWS (product / comment))

The User-Agent field-value consists of one or more product identifiers, each followed by zero or more comments (Section 3.2 of [Part1]), which together identify the user agent software and its significant subproducts. By convention, the product identifiers are listed in decreasing order of their significance for identifying the user agent software. Each product identifier consists of a name and optional version.

product = token ["/" product-version]
product-version = token

A sender **SHOULD** limit generated product identifiers to what is necessary to identify the product; a sender **MUST NOT** generate advertising or other non-essential information within the product identifier. A sender **SHOULD NOT** generate information in product-version that is not a version identifier (i.e., successive versions of the same product name ought to only differ in the product-version portion of the product identifier).

Example:

User-Agent: CERN-LineMode/2.15 libwww/2.17b3

A user agent **SHOULD NOT** generate a User-Agent field containing needlessly fine-grained detail and **SHOULD** limit the addition of subproducts by third parties. Overly long and detailed User-Agent field values increase request latency and the risk of a user being identified against their wishes ("fingerprinting").

Likewise, implementations are encouraged not to use the product tokens of other implementations in order to declare compatibility with them, as this circumvents the purpose of the field. If a user agent masquerades as a different user agent, recipients can assume that the user intentionally desires to see responses tailored for that identified user agent, even if they might not work as well for the actual user agent being used.

6. Response Status Codes

The status-code element is a 3-digit integer code giving the result of the attempt to understand and satisfy the request.

HTTP status codes are extensible. HTTP clients are not required to understand the meaning of all registered status codes, though such understanding is obviously desirable. However, a client **MUST** understand the class of any status code, as indicated by the first digit, and treat an unrecognized status code as being equivalent to the x00 status code of that class, with the exception that a recipient **MUST NOT** cache a response with an unrecognized status code.

For example, if an unrecognized status code of 471 is received by a client, the client can assume that there was something wrong with its request and treat the response as if it had received a 400 status code. The response message will usually contain a representation that explains the status.

The first digit of the status-code defines the class of response. The last two digits do not have any categorization role. There are 5 values for the first digit:

- o 1xx (Informational): The request was received, continuing process
- o 2xx (Successful): The request was successfully received, understood, and accepted
- o 3xx (Redirection): Further action needs to be taken in order to complete the request
- o 4xx (Client Error): The request contains bad syntax or cannot be fulfilled
- o 5xx (Server Error): The server failed to fulfill an apparently valid request

6.1. Overview of Status Codes

The status codes listed below are defined in this specification, Section 4 of [Part4], Section 4 of [Part5], and Section 3 of [Part7]. The reason phrases listed here are only recommendations -- they can be replaced by local equivalents without affecting the protocol.

Responses with status codes that are defined as cacheable by default (e.g., 200, 203, 204, 206, 300, 301, 404, 405, 410, 414, 501 in this specification) can be reused by a cache with heuristic expiration unless otherwise indicated by the method definition or explicit cache

controls [Part6]; all other status codes are not cacheable by default.

| code | reason-phrase | Defined in... |
|------|-------------------------------|------------------------|
| 100 | Continue | Section 6.2.1 |
| 101 | Switching Protocols | Section 6.2.2 |
| 200 | OK | Section 6.3.1 |
| 201 | Created | Section 6.3.2 |
| 202 | Accepted | Section 6.3.3 |
| 203 | Non-Authoritative Information | Section 6.3.4 |
| 204 | No Content | Section 6.3.5 |
| 205 | Reset Content | Section 6.3.6 |
| 206 | Partial Content | Section 4.1 of [Part5] |
| 300 | Multiple Choices | Section 6.4.1 |
| 301 | Moved Permanently | Section 6.4.2 |
| 302 | Found | Section 6.4.3 |
| 303 | See Other | Section 6.4.4 |
| 304 | Not Modified | Section 4.1 of [Part4] |
| 305 | Use Proxy | Section 6.4.5 |
| 307 | Temporary Redirect | Section 6.4.7 |
| 400 | Bad Request | Section 6.5.1 |
| 401 | Unauthorized | Section 3.1 of [Part7] |
| 402 | Payment Required | Section 6.5.2 |
| 403 | Forbidden | Section 6.5.3 |
| 404 | Not Found | Section 6.5.4 |
| 405 | Method Not Allowed | Section 6.5.5 |
| 406 | Not Acceptable | Section 6.5.6 |
| 407 | Proxy Authentication Required | Section 3.2 of [Part7] |
| 408 | Request Time-out | Section 6.5.7 |
| 409 | Conflict | Section 6.5.8 |
| 410 | Gone | Section 6.5.9 |
| 411 | Length Required | Section 6.5.10 |
| 412 | Precondition Failed | Section 4.2 of [Part4] |
| 413 | Payload Too Large | Section 6.5.11 |
| 414 | URI Too Long | Section 6.5.12 |
| 415 | Unsupported Media Type | Section 6.5.13 |
| 416 | Range Not Satisfiable | Section 4.4 of [Part5] |
| 417 | Expectation Failed | Section 6.5.14 |
| 426 | Upgrade Required | Section 6.5.15 |
| 500 | Internal Server Error | Section 6.6.1 |
| 501 | Not Implemented | Section 6.6.2 |
| 502 | Bad Gateway | Section 6.6.3 |
| 503 | Service Unavailable | Section 6.6.4 |
| 504 | Gateway Time-out | Section 6.6.5 |
| 505 | HTTP Version Not Supported | Section 6.6.6 |

Note that this list is not exhaustive -- it does not include extension status codes defined in other specifications. The complete list of status codes is maintained by IANA. See Section 8.2 for details.

6.2. Informational 1xx

The 1xx (Informational) class of status code indicates an interim response for communicating connection status or request progress prior to completing the requested action and sending a final response. All 1xx responses consist of only the status-line and optional header fields, and thus are terminated by the empty line at the end of the header section. Since HTTP/1.0 did not define any 1xx status codes, a server **MUST NOT** send a 1xx response to an HTTP/1.0 client.

A client **MUST** be able to parse one or more 1xx responses received prior to a final response, even if the client does not expect one. A user agent **MAY** ignore unexpected 1xx responses.

A proxy **MUST** forward 1xx responses unless the proxy itself requested the generation of the 1xx response. For example, if a proxy adds an "Expect: 100-continue" field when it forwards a request, then it need not forward the corresponding 100 (Continue) response(s).

6.2.1. 100 Continue

The 100 (Continue) status code indicates that the initial part of a request has been received and has not yet been rejected by the server. The server intends to send a final response after the request has been fully received and acted upon.

When the request contains an Expect header field that includes a 100-continue expectation, the 100 response indicates that the server wishes to receive the request payload body, as described in Section 5.1.1. The client ought to continue sending the request and discard the 100 response.

If the request did not contain an Expect header field containing the 100-continue expectation, the client can simply discard this interim response.

6.2.2. 101 Switching Protocols

The 101 (Switching Protocols) status code indicates that the server understands and is willing to comply with the client's request, via the Upgrade header field (Section 6.7 of [Part1]), for a change in the application protocol being used on this connection. The server

MUST generate an Upgrade header field in the response that indicates which protocol(s) will be switched to immediately after the empty line that terminates the 101 response.

It is assumed that the server will only agree to switch protocols when it is advantageous to do so. For example, switching to a newer version of HTTP might be advantageous over older versions, and switching to a real-time, synchronous protocol might be advantageous when delivering resources that use such features.

6.3. Successful 2xx

The 2xx (Successful) class of status code indicates that the client's request was successfully received, understood, and accepted.

6.3.1. 200 OK

The 200 (OK) status code indicates that the request has succeeded. The payload sent in a 200 response depends on the request method. For the methods defined by this specification, the intended meaning of the payload can be summarized as:

GET a representation of the target resource;

HEAD the same representation as GET, but without the representation data;

POST a representation of the status of, or results obtained from, the action;

PUT, DELETE a representation of the status of the action;

OPTIONS a representation of the communications options;

TRACE a representation of the request message as received by the end server.

Aside from responses to CONNECT, a 200 response always has a payload, though an origin server MAY generate a payload body of zero length. If no payload is desired, an origin server ought to send 204 (No Content) instead. For CONNECT, no payload is allowed because the successful result is a tunnel, which begins immediately after the 200 response header section.

A 200 response is cacheable by default; i.e., unless otherwise indicated by the method definition or explicit cache controls (see Section 4.2.2 of [Part6]).

6.3.2. 201 Created

The 201 (Created) status code indicates that the request has been fulfilled and has resulted in one or more new resources being created. The primary resource created by the request is identified by either a Location header field in the response or, if no Location field is received, by the effective request URI.

The 201 response payload typically describes and links to the resource(s) created. See Section 7.2 for a discussion of the meaning and purpose of validator header fields, such as ETag and Last-Modified, in a 201 response.

6.3.3. 202 Accepted

The 202 (Accepted) status code indicates that the request has been accepted for processing, but the processing has not been completed. The request might or might not eventually be acted upon, as it might be disallowed when processing actually takes place. There is no facility in HTTP for re-sending a status code from an asynchronous operation.

The 202 response is intentionally non-committal. Its purpose is to allow a server to accept a request for some other process (perhaps a batch-oriented process that is only run once per day) without requiring that the user agent's connection to the server persist until the process is completed. The representation sent with this response ought to describe the request's current status and point to (or embed) a status monitor that can provide the user with an estimate of when the request will be fulfilled.

6.3.4. 203 Non-Authoritative Information

The 203 (Non-Authoritative Information) status code indicates that the request was successful but the enclosed payload has been modified from that of the origin server's 200 (OK) response by a transforming proxy (Section 5.7.2 of [Part1]). This status code allows the proxy to notify recipients when a transformation has been applied, since that knowledge might impact later decisions regarding the content. For example, future cache validation requests for the content might only be applicable along the same request path (through the same proxies).

The 203 response is similar to the Warning code of 214 Transformation Applied (Section 5.5 of [Part6]), which has the advantage of being applicable to responses with any status code.

A 203 response is cacheable by default; i.e., unless otherwise

indicated by the method definition or explicit cache controls (see Section 4.2.2 of [Part6]).

6.3.5. 204 No Content

The 204 (No Content) status code indicates that the server has successfully fulfilled the request and that there is no additional content to send in the response payload body. Metadata in the response header fields refer to the target resource and its selected representation after the requested action was applied.

For example, if a 204 status code is received in response to a PUT request and the response contains an ETag header field, then the PUT was successful and the ETag field-value contains the entity-tag for the new representation of that target resource.

The 204 response allows a server to indicate that the action has been successfully applied to the target resource, while implying that the user agent does not need to traverse away from its current "document view" (if any). The server assumes that the user agent will provide some indication of the success to its user, in accord with its own interface, and apply any new or updated metadata in the response to its active representation.

For example, a 204 status code is commonly used with document editing interfaces corresponding to a "save" action, such that the document being saved remains available to the user for editing. It is also frequently used with interfaces that expect automated data transfers to be prevalent, such as within distributed version control systems.

A 204 response is terminated by the first empty line after the header fields because it cannot contain a message body.

A 204 response is cacheable by default; i.e., unless otherwise indicated by the method definition or explicit cache controls (see Section 4.2.2 of [Part6]).

6.3.6. 205 Reset Content

The 205 (Reset Content) status code indicates that the server has fulfilled the request and desires that the user agent reset the "document view", which caused the request to be sent, to its original state as received from the origin server.

This response is intended to support a common data entry use case where the user receives content that supports data entry (a form, notepad, canvas, etc.), enters or manipulates data in that space, causes the entered data to be submitted in a request, and then the

data entry mechanism is reset for the next entry so that the user can easily initiate another input action.

Since the 205 status code implies that no additional content will be provided, a server **MUST NOT** generate a payload in a 205 response. In other words, a server **MUST** do one of the following for a 205 response: a) indicate a zero-length body for the response by including a Content-Length header field with a value of 0; b) indicate a zero-length payload for the response by including a Transfer-Encoding header field with a value of chunked and a message body consisting of a single chunk of zero-length; or, c) close the connection immediately after sending the blank line terminating the header section.

6.4. Redirection 3xx

The 3xx (Redirection) class of status code indicates that further action needs to be taken by the user agent in order to fulfill the request. If a Location header field (Section 7.1.2) is provided, the user agent **MAY** automatically redirect its request to the URI referenced by the Location field value, even if the specific status code is not understood. Automatic redirection needs to be done with care for methods not known to be safe, as defined in Section 4.2.1, since the user might not wish to redirect an unsafe request.

There are several types of redirects:

1. Redirects that indicate the resource might be available at a different URI, as provided by the Location field, as in the status codes 301 (Moved Permanently), 302 (Found), and 307 (Temporary Redirect).
2. Redirection that offers a choice of matching resources, each capable of representing the original request target, as in the 300 (Multiple Choices) status code.
3. Redirection to a different resource, identified by the Location field, that can represent an indirect response to the request, as in the 303 (See Other) status code.
4. Redirection to a previously cached result, as in the 304 (Not Modified) status code.

Note: In HTTP/1.0, the status codes 301 (Moved Permanently) and 302 (Found) were defined for the first type of redirect ([RFC1945], Section 9.3). Early user agents split on whether the method applied to the redirect target would be the same as the original request or would be rewritten as GET. Although HTTP

originally defined the former semantics for 301 and 302 (to match its original implementation at CERN), and defined 303 (See Other) to match the latter semantics, prevailing practice gradually converged on the latter semantics for 301 and 302 as well. The first revision of HTTP/1.1 added 307 (Temporary Redirect) to indicate the former semantics without being impacted by divergent practice. Over 10 years later, most user agents still do method rewriting for 301 and 302; therefore, this specification makes that behavior conformant when the original request is POST.

A client SHOULD detect and intervene in cyclical redirections (i.e., "infinite" redirection loops).

Note: An earlier version of this specification recommended a maximum of five redirections ([RFC2068], Section 10.3). Content developers need to be aware that some clients might implement such a fixed limitation.

6.4.1. 300 Multiple Choices

The 300 (Multiple Choices) status code indicates that the target resource has more than one representation, each with its own more specific identifier, and information about the alternatives is being provided so that the user (or user agent) can select a preferred representation by redirecting its request to one or more of those identifiers. In other words, the server desires that the user agent engage in reactive negotiation to select the most appropriate representation(s) for its needs (Section 3.4).

If the server has a preferred choice, the server SHOULD generate a Location header field containing a preferred choice's URI reference. The user agent MAY use the Location field value for automatic redirection.

For request methods other than HEAD, the server SHOULD generate a payload in the 300 response containing a list of representation metadata and URI reference(s) from which the user or user agent can choose the one most preferred. The user agent MAY make a selection from that list automatically if it understands the provided media type. A specific format for automatic selection is not defined by this specification because HTTP tries to remain orthogonal to the definition of its payloads. In practice, the representation is provided in some easily parsed format believed to be acceptable to the user agent, as determined by shared design or content negotiation, or in some commonly accepted hypertext format.

A 300 response is cacheable by default; i.e., unless otherwise indicated by the method definition or explicit cache controls (see

Section 4.2.2 of [Part6]).

Note: The original proposal for 300 defined the URI header field as providing a list of alternative representations, such that it would be usable for 200, 300, and 406 responses and be transferred in responses to the HEAD method. However, lack of deployment and disagreement over syntax led to both URI and Alternates (a subsequent proposal) being dropped from this specification. It is possible to communicate the list using a set of Link header fields [RFC5988], each with a relationship of "alternate", though deployment is a chicken-and-egg problem.

6.4.2. 301 Moved Permanently

The 301 (Moved Permanently) status code indicates that the target resource has been assigned a new permanent URI and any future references to this resource ought to use one of the enclosed URIs. Clients with link editing capabilities ought to automatically re-link references to the effective request URI to one or more of the new references sent by the server, where possible.

The server SHOULD generate a Location header field in the response containing a preferred URI reference for the new permanent URI. The user agent MAY use the Location field value for automatic redirection. The server's response payload usually contains a short hypertext note with a hyperlink to the new URI(s).

Note: For historical reasons, a user agent MAY change the request method from POST to GET for the subsequent request. If this behavior is undesired, the 307 (Temporary Redirect) status code can be used instead.

A 301 response is cacheable by default; i.e., unless otherwise indicated by the method definition or explicit cache controls (see Section 4.2.2 of [Part6]).

6.4.3. 302 Found

The 302 (Found) status code indicates that the target resource resides temporarily under a different URI. Since the redirection might be altered on occasion, the client ought to continue to use the effective request URI for future requests.

The server SHOULD generate a Location header field in the response containing a URI reference for the different URI. The user agent MAY use the Location field value for automatic redirection. The server's response payload usually contains a short hypertext note with a hyperlink to the different URI(s).

Note: For historical reasons, a user agent MAY change the request method from POST to GET for the subsequent request. If this behavior is undesired, the 307 (Temporary Redirect) status code can be used instead.

6.4.4. 303 See Other

The 303 (See Other) status code indicates that the server is redirecting the user agent to a different resource, as indicated by a URI in the Location header field, which is intended to provide an indirect response to the original request. A user agent can perform a retrieval request targeting that URI (a GET or HEAD request if using HTTP), which might also be redirected, and present the eventual result as an answer to the original request. Note that the new URI in the Location header field is not considered equivalent to the effective request URI.

This status code is applicable to any HTTP method. It is primarily used to allow the output of a POST action to redirect the user agent to a selected resource, since doing so provides the information corresponding to the POST response in a form that can be separately identified, bookmarked, and cached independent of the original request.

A 303 response to a GET request indicates that the origin server does not have a representation of the target resource that can be transferred by the server over HTTP. However, the Location field value refers to a resource that is descriptive of the target resource, such that making a retrieval request on that other resource might result in a representation that is useful to recipients without implying that it represents the original target resource. Note that answers to the questions of what can be represented, what representations are adequate, and what might be a useful description are outside the scope of HTTP.

Except for responses to a HEAD request, the representation of a 303 response ought to contain a short hypertext note with a hyperlink to the same URI reference provided in the Location header field.

6.4.5. 305 Use Proxy

The 305 (Use Proxy) status code was defined in a previous version of this specification and is now deprecated (Appendix B).

6.4.6. 306 (Unused)

The 306 status code was defined in a previous version of this specification, is no longer used, and the code is reserved.

6.4.7. 307 Temporary Redirect

The 307 (Temporary Redirect) status code indicates that the target resource resides temporarily under a different URI and the user agent **MUST NOT** change the request method if it performs an automatic redirection to that URI. Since the redirection can change over time, the client ought to continue using the original effective request URI for future requests.

The server **SHOULD** generate a Location header field in the response containing a URI reference for the different URI. The user agent **MAY** use the Location field value for automatic redirection. The server's response payload usually contains a short hypertext note with a hyperlink to the different URI(s).

Note: This status code is similar to 302 (Found), except that it does not allow changing the request method from POST to GET. This specification defines no equivalent counterpart for 301 (Moved Permanently) ([status-308], however, defines the status code 308 (Permanent Redirect) for this purpose).

6.5. Client Error 4xx

The 4xx (Client Error) class of status code indicates that the client seems to have erred. Except when responding to a HEAD request, the server **SHOULD** send a representation containing an explanation of the error situation, and whether it is a temporary or permanent condition. These status codes are applicable to any request method. User agents **SHOULD** display any included representation to the user.

6.5.1. 400 Bad Request

The 400 (Bad Request) status code indicates that the server cannot or will not process the request due to something which is perceived to be a client error (e.g., malformed request syntax, invalid request message framing, or deceptive request routing).

6.5.2. 402 Payment Required

The 402 (Payment Required) status code is reserved for future use.

6.5.3. 403 Forbidden

The 403 (Forbidden) status code indicates that the server understood the request but refuses to authorize it. A server that wishes to make public why the request has been forbidden can describe that reason in the response payload (if any).

If authentication credentials were provided in the request, the server considers them insufficient to grant access. The client SHOULD NOT automatically repeat the request with the same credentials. The client MAY repeat the request with new or different credentials. However, a request might be forbidden for reasons unrelated to the credentials.

An origin server that wishes to "hide" the current existence of a forbidden target resource MAY instead respond with a status code of 404 (Not Found).

6.5.4. 404 Not Found

The 404 (Not Found) status code indicates that the origin server did not find a current representation for the target resource or is not willing to disclose that one exists. A 404 status code does not indicate whether this lack of representation is temporary or permanent; the 410 (Gone) status code is preferred over 404 if the origin server knows, presumably through some configurable means, that the condition is likely to be permanent.

A 404 response is cacheable by default; i.e., unless otherwise indicated by the method definition or explicit cache controls (see Section 4.2.2 of [Part6]).

6.5.5. 405 Method Not Allowed

The 405 (Method Not Allowed) status code indicates that the method received in the request-line is known by the origin server but not supported by the target resource. The origin server MUST generate an Allow header field in a 405 response containing a list of the target resource's currently supported methods.

A 405 response is cacheable by default; i.e., unless otherwise indicated by the method definition or explicit cache controls (see Section 4.2.2 of [Part6]).

6.5.6. 406 Not Acceptable

The 406 (Not Acceptable) status code indicates that the target resource does not have a current representation that would be acceptable to the user agent, according to the proactive negotiation header fields received in the request (Section 5.3), and the server is unwilling to supply a default representation.

The server SHOULD generate a payload containing a list of available representation characteristics and corresponding resource identifiers from which the user or user agent can choose the one most

appropriate. A user agent MAY automatically select the most appropriate choice from that list. However, this specification does not define any standard for such automatic selection, as described in Section 6.4.1.

6.5.7. 408 Request Timeout

The 408 (Request Timeout) status code indicates that the server did not receive a complete request message within the time that it was prepared to wait. A server SHOULD send the close connection option (Section 6.1 of [Part1]) in the response, since 408 implies that the server has decided to close the connection rather than continue waiting. If the client has an outstanding request in transit, the client MAY repeat that request on a new connection.

6.5.8. 409 Conflict

The 409 (Conflict) status code indicates that the request could not be completed due to a conflict with the current state of the target resource. This code is used in situations where the user might be able to resolve the conflict and resubmit the request. The server SHOULD generate a payload that includes enough information for a user to recognize the source of the conflict.

Conflicts are most likely to occur in response to a PUT request. For example, if versioning were being used and the representation being PUT included changes to a resource that conflict with those made by an earlier (third-party) request, the origin server might use a 409 response to indicate that it can't complete the request. In this case, the response representation would likely contain information useful for merging the differences based on the revision history.

6.5.9. 410 Gone

The 410 (Gone) status code indicates that access to the target resource is no longer available at the origin server and that this condition is likely to be permanent. If the origin server does not know, or has no facility to determine, whether or not the condition is permanent, the status code 404 (Not Found) ought to be used instead.

The 410 response is primarily intended to assist the task of web maintenance by notifying the recipient that the resource is intentionally unavailable and that the server owners desire that remote links to that resource be removed. Such an event is common for limited-time, promotional services and for resources belonging to individuals no longer associated with the origin server's site. It is not necessary to mark all permanently unavailable resources as

"gone" or to keep the mark for any length of time -- that is left to the discretion of the server owner.

A 410 response is cacheable by default; i.e., unless otherwise indicated by the method definition or explicit cache controls (see Section 4.2.2 of [Part6]).

6.5.10. 411 Length Required

The 411 (Length Required) status code indicates that the server refuses to accept the request without a defined Content-Length (Section 3.3.2 of [Part1]). The client MAY repeat the request if it adds a valid Content-Length header field containing the length of the message body in the request message.

6.5.11. 413 Payload Too Large

The 413 (Payload Too Large) status code indicates that the server is refusing to process a request because the request payload is larger than the server is willing or able to process. The server MAY close the connection to prevent the client from continuing the request.

If the condition is temporary, the server SHOULD generate a Retry-After header field to indicate that it is temporary and after what time the client MAY try again.

6.5.12. 414 URI Too Long

The 414 (URI Too Long) status code indicates that the server is refusing to service the request because the request-target (Section 5.3 of [Part1]) is longer than the server is willing to interpret. This rare condition is only likely to occur when a client has improperly converted a POST request to a GET request with long query information, when the client has descended into a "black hole" of redirection (e.g., a redirected URI prefix that points to a suffix of itself), or when the server is under attack by a client attempting to exploit potential security holes.

A 414 response is cacheable by default; i.e., unless otherwise indicated by the method definition or explicit cache controls (see Section 4.2.2 of [Part6]).

6.5.13. 415 Unsupported Media Type

The 415 (Unsupported Media Type) status code indicates that the origin server is refusing to service the request because the payload is in a format not supported by this method on the target resource. The format problem might be due to the request's indicated Content-

Type or Content-Encoding, or as a result of inspecting the data directly.

6.5.14. 417 Expectation Failed

The 417 (Expectation Failed) status code indicates that the expectation given in the request's Expect header field (Section 5.1.1) could not be met by at least one of the inbound servers.

6.5.15. 426 Upgrade Required

The 426 (Upgrade Required) status code indicates that the server refuses to perform the request using the current protocol but might be willing to do so after the client upgrades to a different protocol. The server **MUST** send an Upgrade header field in a 426 response to indicate the required protocol(s) (Section 6.7 of [Part1]).

Example:

```
HTTP/1.1 426 Upgrade Required
Upgrade: HTTP/3.0
Connection: Upgrade
Content-Length: 53
Content-Type: text/plain
```

This service requires use of the HTTP/3.0 protocol.

6.6. Server Error 5xx

The 5xx (Server Error) class of status code indicates that the server is aware that it has erred or is incapable of performing the requested method. Except when responding to a HEAD request, the server **SHOULD** send a representation containing an explanation of the error situation, and whether it is a temporary or permanent condition. A user agent **SHOULD** display any included representation to the user. These response codes are applicable to any request method.

6.6.1. 500 Internal Server Error

The 500 (Internal Server Error) status code indicates that the server encountered an unexpected condition that prevented it from fulfilling the request.

6.6.2. 501 Not Implemented

The 501 (Not Implemented) status code indicates that the server does not support the functionality required to fulfill the request. This is the appropriate response when the server does not recognize the request method and is not capable of supporting it for any resource.

A 501 response is cacheable by default; i.e., unless otherwise indicated by the method definition or explicit cache controls (see Section 4.2.2 of [Part6]).

6.6.3. 502 Bad Gateway

The 502 (Bad Gateway) status code indicates that the server, while acting as a gateway or proxy, received an invalid response from an inbound server it accessed while attempting to fulfill the request.

6.6.4. 503 Service Unavailable

The 503 (Service Unavailable) status code indicates that the server is currently unable to handle the request due to a temporary overload or scheduled maintenance, which will likely be alleviated after some delay. The server MAY send a Retry-After header field (Section 7.1.3) to suggest an appropriate amount of time for the client to wait before retrying the request.

Note: The existence of the 503 status code does not imply that a server has to use it when becoming overloaded. Some servers might simply refuse the connection.

6.6.5. 504 Gateway Timeout

The 504 (Gateway Timeout) status code indicates that the server, while acting as a gateway or proxy, did not receive a timely response from an upstream server it needed to access in order to complete the request.

6.6.6. 505 HTTP Version Not Supported

The 505 (HTTP Version Not Supported) status code indicates that the server does not support, or refuses to support, the major version of HTTP that was used in the request message. The server is indicating that it is unable or unwilling to complete the request using the same major version as the client, as described in Section 2.6 of [Part1], other than with this error message. The server SHOULD generate a representation for the 505 response that describes why that version is not supported and what other protocols are supported by that server.

7. Response Header Fields

The response header fields allow the server to pass additional information about the response beyond what is placed in the status-line. These header fields give information about the server, about further access to the target resource, or about related resources.

Although each response header field has a defined meaning, in general, the precise semantics might be further refined by the semantics of the request method and/or response status code.

7.1. Control Data

Response header fields can supply control data that supplements the status code, directs caching, or instructs the client where to go next.

| Header Field Name | Defined in... |
|-------------------|------------------------|
| Age | Section 5.1 of [Part6] |
| Cache-Control | Section 5.2 of [Part6] |
| Expires | Section 5.3 of [Part6] |
| Date | Section 7.1.1.2 |
| Location | Section 7.1.2 |
| Retry-After | Section 7.1.3 |
| Vary | Section 7.1.4 |
| Warning | Section 5.5 of [Part6] |

7.1.1. Origination Date

7.1.1.1. Date/Time Formats

Prior to 1995, there were three different formats commonly used by servers to communicate timestamps. For compatibility with old implementations, all three are defined here. The preferred format is a fixed-length and single-zone subset of the date and time specification used by the Internet Message Format [RFC5322].

HTTP-date = IMF-fixdate / obs-date

An example of the preferred format is

Sun, 06 Nov 1994 08:49:37 GMT ; IMF-fixdate

Examples of the two obsolete formats are

```
Sunday, 06-Nov-94 08:49:37 GMT    ; obsolete RFC 850 format
Sun Nov  6 08:49:37 1994         ; ANSI C's asctime() format
```

A recipient that parses a timestamp value in an HTTP header field MUST accept all three HTTP-date formats. When a sender generates a header field that contains one or more timestamps defined as HTTP-date, the sender MUST generate those timestamps in the IMF-fixdate format.

An HTTP-date value represents time as an instance of Coordinated Universal Time (UTC). The first two formats indicate UTC by the three-letter abbreviation for Greenwich Mean Time, "GMT", a predecessor of the UTC name; values in the asctime format are assumed to be in UTC. A sender that generates HTTP-date values from a local clock ought to use NTP ([RFC5905]) or some similar protocol to synchronize its clock to UTC.

Preferred format:

IMF-fixdate = day-name "," SP date1 SP time-of-day SP GMT
; fixed length/zone/capitalization subset of the format
; defined in Section 3.3 of [RFC5322]

day-name = %x4D.6F.6E ; "Mon", case-sensitive
/ %x54.75.65 ; "Tue", case-sensitive
/ %x57.65.64 ; "Wed", case-sensitive
/ %x54.68.75 ; "Thu", case-sensitive
/ %x46.72.69 ; "Fri", case-sensitive
/ %x53.61.74 ; "Sat", case-sensitive
/ %x53.75.6E ; "Sun", case-sensitive

date1 = day SP month SP year
; e.g., 02 Jun 1982

day = 2DIGIT
month = %x4A.61.6E ; "Jan", case-sensitive
/ %x46.65.62 ; "Feb", case-sensitive
/ %x4D.61.72 ; "Mar", case-sensitive
/ %x41.70.72 ; "Apr", case-sensitive
/ %x4D.61.79 ; "May", case-sensitive
/ %x4A.75.6E ; "Jun", case-sensitive
/ %x4A.75.6C ; "Jul", case-sensitive
/ %x41.75.67 ; "Aug", case-sensitive
/ %x53.65.70 ; "Sep", case-sensitive
/ %x4F.63.74 ; "Oct", case-sensitive
/ %x4E.6F.76 ; "Nov", case-sensitive
/ %x44.65.63 ; "Dec", case-sensitive

year = 4DIGIT

GMT = %x47.4D.54 ; "GMT", case-sensitive

time-of-day = hour ":" minute ":" second
; 00:00:00 - 23:59:60 (leap second)

hour = 2DIGIT

minute = 2DIGIT

second = 2DIGIT

Obsolete formats:

obs-date = rfc850-date / asctime-date

```

rfc850-date  = day-name-1 "," SP date2 SP time-of-day SP GMT
date2        = day "-" month "-" 2DIGIT
              ; e.g., 02-Jun-82

day-name-1    = %x4D.6F.6E.64.61.79      ; "Monday", case-sensitive
              / %x54.75.65.73.64.61.79    ; "Tuesday", case-sensitive
              / %x57.65.64.6E.65.73.64.61.79 ; "Wednesday", case-sensitive
              / %x54.68.75.72.73.64.61.79    ; "Thursday", case-sensitive
              / %x46.72.69.64.61.79          ; "Friday", case-sensitive
              / %x53.61.74.75.72.64.61.79    ; "Saturday", case-sensitive
              / %x53.75.6E.64.61.79          ; "Sunday", case-sensitive

asctime-date  = day-name SP date3 SP time-of-day SP year
date3        = month SP ( 2DIGIT / ( SP 1DIGIT ) )
              ; e.g., Jun  2

```

HTTP-date is case sensitive. A sender MUST NOT generate additional whitespace in an HTTP-date beyond that specifically included as SP in the grammar. The semantics of day-name, day, month, year, and time-of-day are the same as those defined for the Internet Message Format constructs with the corresponding name ([RFC5322], Section 3.3).

Recipients of a timestamp value in rfc850-date format, which uses a two-digit year, MUST interpret a timestamp that appears to be more than 50 years in the future as representing the most recent year in the past that had the same last two digits.

Recipients of timestamp values are encouraged to be robust in parsing timestamps unless otherwise restricted by the field definition. For example, messages are occasionally forwarded over HTTP from a non-HTTP source that might generate any of the date and time specifications defined by the Internet Message Format.

Note: HTTP requirements for the date/time stamp format apply only to their usage within the protocol stream. Implementations are not required to use these formats for user presentation, request logging, etc.

7.1.1.2. Date

The "Date" header field represents the date and time at which the message was originated, having the same semantics as the Origination Date Field (orig-date) defined in Section 3.6.1 of [RFC5322]. The field value is an HTTP-date, as defined in Section 7.1.1.1.

Date = HTTP-date

An example is

Date: Tue, 15 Nov 1994 08:12:31 GMT

When a Date header field is generated, the sender SHOULD generate its field value as the best available approximation of the date and time of message generation. In theory, the date ought to represent the moment just before the payload is generated. In practice, the date can be generated at any time during message origination.

An origin server MUST NOT send a Date header field if it does not have a clock capable of providing a reasonable approximation of the current instance in Coordinated Universal Time. An origin server MAY send a Date header field if the response is in the 1xx (Informational) or 5xx (Server Error) class of status codes. An origin server MUST send a Date header field in all other cases.

A recipient with a clock that receives a response message without a Date header field MUST record the time it was received and append a corresponding Date header field to the message's header section if it is cached or forwarded downstream.

A user agent MAY send a Date header field in a request, though generally will not do so unless it is believed to convey useful information to the server. For example, custom applications of HTTP might convey a Date if the server is expected to adjust its interpretation of the user's request based on differences between the user agent and server clocks.

7.1.2. Location

The "Location" header field is used in some responses to refer to a specific resource in relation to the response. The type of relationship is defined by the combination of request method and status code semantics.

Location = URI-reference

The field value consists of a single URI-reference. When it has the form of a relative reference ([RFC3986], Section 4.2), the final value is computed by resolving it against the effective request URI ([RFC3986], Section 5).

For 201 (Created) responses, the Location value refers to the primary resource created by the request. For 3xx (Redirection) responses, the Location value refers to the preferred target resource for automatically redirecting the request.

If the Location value provided in a 3xx (Redirection) does not have a fragment component, a user agent MUST process the redirection as if the value inherits the fragment component of the URI reference used to generate the request target (i.e., the redirection inherits the original reference's fragment, if any).

For example, a GET request generated for the URI reference "http://www.example.org/~tim" might result in a 303 (See Other) response containing the header field:

```
Location: /People.html#tim
```

which suggests that the user agent redirect to "http://www.example.org/People.html#tim"

Likewise, a GET request generated for the URI reference "http://www.example.org/index.html#larry" might result in a 301 (Moved Permanently) response containing the header field:

```
Location: http://www.example.net/index.html
```

which suggests that the user agent redirect to "http://www.example.net/index.html#larry", preserving the original fragment identifier.

There are circumstances in which a fragment identifier in a Location value would not be appropriate. For example, the Location header field in a 201 (Created) response is supposed to provide a URI that is specific to the created resource.

Note: Some recipients attempt to recover from Location fields that are not valid URI references. This specification does not mandate or define such processing, but does allow it for the sake of robustness.

Note: The Content-Location header field (Section 3.1.4.2) differs from Location in that the Content-Location refers to the most specific resource corresponding to the enclosed representation. It is therefore possible for a response to contain both the Location and Content-Location header fields.

7.1.3. Retry-After

Servers send the "Retry-After" header field to indicate how long the user agent ought to wait before making a follow-up request. When sent with a 503 (Service Unavailable) response, Retry-After indicates how long the service is expected to be unavailable to the client. When sent with any 3xx (Redirection) response, Retry-After indicates

the minimum time that the user agent is asked to wait before issuing the redirected request.

The value of this field can be either an HTTP-date or a number of seconds to delay after the response is received.

Retry-After = HTTP-date / delay-seconds

A delay-seconds value is a non-negative decimal integer, representing time in seconds.

delay-seconds = 1*DIGIT

Two examples of its use are

Retry-After: Fri, 31 Dec 1999 23:59:59 GMT
Retry-After: 120

In the latter example, the delay is 2 minutes.

7.1.4. Vary

The "Vary" header field in a response describes what parts of a request message, aside from the method, Host header field, and request target, might influence the origin server's process for selecting and representing this response. The value consists of either a single asterisk ("*") or a list of header field names (case-insensitive).

Vary = "*" / 1#field-name

A Vary field value of "*" signals that anything about the request might play a role in selecting the response representation, possibly including elements outside the message syntax (e.g., the client's network address). A recipient will not be able to determine whether this response is appropriate for a later request without forwarding the request to the origin server. A proxy MUST NOT generate a Vary field with a "*" value.

A Vary field value consisting of a comma-separated list of names indicates that the named request header fields, known as the selecting header fields, might have a role in selecting the representation. The potential selecting header fields are not limited to those defined by this specification.

For example, a response that contains

Vary: accept-encoding, accept-language

indicates that the origin server might have used the request's Accept-Encoding and Accept-Language fields (or lack thereof) as determining factors while choosing the content for this response.

An origin server might send Vary with a list of fields for two purposes:

1. To inform cache recipients that they **MUST NOT** use this response to satisfy a later request unless the later request has the same values for the listed fields as the original request (Section 4.1 of [Part6]). In other words, Vary expands the cache key required to match a new request to the stored cache entry.
2. To inform user agent recipients that this response is subject to content negotiation (Section 5.3) and that a different representation might be sent in a subsequent request if additional parameters are provided in the listed header fields (proactive negotiation).

An origin server **SHOULD** send a Vary header field when its algorithm for selecting a representation varies based on aspects of the request message other than the method and request target, unless the variance cannot be crossed or the origin server has been deliberately configured to prevent cache transparency. For example, there is no need to send the Authorization field name in Vary because reuse across users is constrained by the field definition (Section 4.2 of [Part7]). Likewise, an origin server might use Cache-Control directives (Section 5.2 of [Part6]) to supplant Vary if it considers the variance less significant than the performance cost of Vary's impact on caching.

7.2. Validator Header Fields

Validator header fields convey metadata about the selected representation (Section 3). In responses to safe requests, validator fields describe the selected representation chosen by the origin server while handling the response. Note that, depending on the status code semantics, the selected representation for a given response is not necessarily the same as the representation enclosed as response payload.

In a successful response to a state-changing request, validator fields describe the new representation that has replaced the prior selected representation as a result of processing the request.

For example, an ETag header field in a 201 response communicates the entity-tag of the newly created resource's representation, so that it can be used in later conditional requests to prevent the "lost update" problem [Part4].

| Header Field Name | Defined in... |
|-------------------|------------------------|
| ETag | Section 2.3 of [Part4] |
| Last-Modified | Section 2.2 of [Part4] |

7.3. Authentication Challenges

Authentication challenges indicate what mechanisms are available for the client to provide authentication credentials in future requests.

| Header Field Name | Defined in... |
|--------------------|------------------------|
| WWW-Authenticate | Section 4.1 of [Part7] |
| Proxy-Authenticate | Section 4.3 of [Part7] |

7.4. Response Context

The remaining response header fields provide more information about the target resource for potential use in later requests.

| Header Field Name | Defined in... |
|-------------------|------------------------|
| Accept-Ranges | Section 2.3 of [Part5] |
| Allow | Section 7.4.1 |
| Server | Section 7.4.2 |

7.4.1. Allow

The "Allow" header field lists the set of methods advertised as supported by the target resource. The purpose of this field is strictly to inform the recipient of valid request methods associated with the resource.

Allow = #method

Example of use:

Allow: GET, HEAD, PUT

The actual set of allowed methods is defined by the origin server at the time of each request. An origin server **MUST** generate an Allow field in a 405 (Method Not Allowed) response and **MAY** do so in any other response. An empty Allow field value indicates that the resource allows no methods, which might occur in a 405 response if the resource has been temporarily disabled by configuration.

A proxy **MUST NOT** modify the Allow header field -- it does not need to understand all of the indicated methods in order to handle them according to the generic message handling rules.

7.4.2. Server

The "Server" header field contains information about the software used by the origin server to handle the request, which is often used by clients to help identify the scope of reported interoperability problems, to work around or tailor requests to avoid particular server limitations, and for analytics regarding server or operating system use. An origin server **MAY** generate a Server field in its responses.

Server = product *(RWS (product / comment))

The Server field-value consists of one or more product identifiers, each followed by zero or more comments (Section 3.2 of [Part1]), which together identify the origin server software and its significant subproducts. By convention, the product identifiers are listed in decreasing order of their significance for identifying the origin server software. Each product identifier consists of a name and optional version, as defined in Section 5.5.3.

Example:

Server: CERN/3.0 libwww/2.17

An origin server **SHOULD NOT** generate a Server field containing needlessly fine-grained detail and **SHOULD** limit the addition of subproducts by third parties. Overly long and detailed Server field values increase response latency and potentially reveal internal implementation details that might make it (slightly) easier for attackers to find and exploit known security holes.

8. IANA Considerations

8.1. Method Registry

The HTTP Method Registry defines the name space for the request method token (Section 4). The method registry will be created and maintained at (the suggested URI)
<<http://www.iana.org/assignments/http-methods>>.

8.1.1. Procedure

HTTP method registrations MUST include the following fields:

- o Method Name (see Section 4)
- o Safe ("yes" or "no", see Section 4.2.1)
- o Idempotent ("yes" or "no", see Section 4.2.2)
- o Pointer to specification text

Values to be added to this name space require IETF Review (see [RFC5226], Section 4.1).

8.1.2. Considerations for New Methods

Standardized methods are generic; that is, they are potentially applicable to any resource, not just one particular media type, kind of resource, or application. As such, it is preferred that new methods be registered in a document that isn't specific to a single application or data format, since orthogonal technologies deserve orthogonal specification.

Since message parsing (Section 3.3 of [Part1]) needs to be independent of method semantics (aside from responses to HEAD), definitions of new methods cannot change the parsing algorithm or prohibit the presence of a message body on either the request or the response message. Definitions of new methods can specify that only a zero-length message body is allowed by requiring a Content-Length header field with a value of "0".

A new method definition needs to indicate whether it is safe (Section 4.2.1), idempotent (Section 4.2.2), cacheable (Section 4.2.3), what semantics are to be associated with the payload body if any is present in the request, and what refinements the method makes to header field or status code semantics. If the new method is cacheable, its definition ought to describe how, and under what conditions, a cache can store a response and use it to satisfy a subsequent request. The new method ought to describe whether it can be made conditional (Section 5.2) and, if so, how a server responds

when the condition is false. Likewise, if the new method might have some use for partial response semantics ([Part5]), it ought to document this too.

Note: Avoid defining a method name that starts with "M-", since that prefix might be misinterpreted as having the semantics assigned to it by [RFC2774].

8.1.3. Registrations

The HTTP Method Registry shall be populated with the registrations below:

| Method | Safe | Idempotent | Reference |
|---------|------|------------|---------------|
| CONNECT | no | no | Section 4.3.6 |
| DELETE | no | yes | Section 4.3.5 |
| GET | yes | yes | Section 4.3.1 |
| HEAD | yes | yes | Section 4.3.2 |
| OPTIONS | yes | yes | Section 4.3.7 |
| POST | no | no | Section 4.3.3 |
| PUT | no | yes | Section 4.3.4 |
| TRACE | yes | yes | Section 4.3.8 |

8.2. Status Code Registry

The HTTP Status Code Registry defines the name space for the response status-code token (Section 6). The status code registry is maintained at <<http://www.iana.org/assignments/http-status-codes>>.

This Section replaces the registration procedure for HTTP Status Codes previously defined in Section 7.1 of [RFC2817].

8.2.1. Procedure

A registration MUST include the following fields:

- o Status Code (3 digits)
- o Short Description
- o Pointer to specification text

Values to be added to the HTTP status code name space require IETF Review (see [RFC5226], Section 4.1).

8.2.2. Considerations for New Status Codes

When it is necessary to express semantics for a response that are not defined by current status codes, a new status code can be registered. Status codes are generic; they are potentially applicable to any resource, not just one particular media type, kind of resource, or application of HTTP. As such, it is preferred that new status codes be registered in a document that isn't specific to a single application.

New status codes are required to fall under one of the categories defined in Section 6. To allow existing parsers to process the response message, new status codes cannot disallow a payload, although they can mandate a zero-length payload body.

Proposals for new status codes that are not yet widely deployed ought to avoid allocating a specific number for the code until there is clear consensus that it will be registered; instead, early drafts can use a notation such as "4NN", or "3N0" .. "3N9", to indicate the class of the proposed status code(s) without consuming a number prematurely.

The definition of a new status code ought to explain the request conditions that would cause a response containing that status code (e.g., combinations of request header fields and/or method(s)) along with any dependencies on response header fields (e.g., what fields are required, what fields can modify the semantics, and what header field semantics are further refined when used with the new status code).

The definition of a new status code ought to specify whether or not it is cacheable. Note that all status codes can be cached if the response they occur in has explicit freshness information; however, status codes that are defined as being cacheable are allowed to be cached without explicit freshness information. Likewise, the definition of a status code can place constraints upon cache behavior. See [Part6] for more information.

Finally, the definition of a new status code ought to indicate whether the payload has any implied association with an identified resource (Section 3.1.4.1).

8.2.3. Registrations

The HTTP Status Code Registry shall be updated with the registrations below:

| Value | Description | Reference |
|-------|-------------------------------|----------------|
| 100 | Continue | Section 6.2.1 |
| 101 | Switching Protocols | Section 6.2.2 |
| 200 | OK | Section 6.3.1 |
| 201 | Created | Section 6.3.2 |
| 202 | Accepted | Section 6.3.3 |
| 203 | Non-Authoritative Information | Section 6.3.4 |
| 204 | No Content | Section 6.3.5 |
| 205 | Reset Content | Section 6.3.6 |
| 300 | Multiple Choices | Section 6.4.1 |
| 301 | Moved Permanently | Section 6.4.2 |
| 302 | Found | Section 6.4.3 |
| 303 | See Other | Section 6.4.4 |
| 305 | Use Proxy | Section 6.4.5 |
| 306 | (Unused) | Section 6.4.6 |
| 307 | Temporary Redirect | Section 6.4.7 |
| 400 | Bad Request | Section 6.5.1 |
| 402 | Payment Required | Section 6.5.2 |
| 403 | Forbidden | Section 6.5.3 |
| 404 | Not Found | Section 6.5.4 |
| 405 | Method Not Allowed | Section 6.5.5 |
| 406 | Not Acceptable | Section 6.5.6 |
| 408 | Request Timeout | Section 6.5.7 |
| 409 | Conflict | Section 6.5.8 |
| 410 | Gone | Section 6.5.9 |
| 411 | Length Required | Section 6.5.10 |
| 413 | Payload Too Large | Section 6.5.11 |
| 414 | URI Too Long | Section 6.5.12 |
| 415 | Unsupported Media Type | Section 6.5.13 |
| 417 | Expectation Failed | Section 6.5.14 |
| 426 | Upgrade Required | Section 6.5.15 |
| 500 | Internal Server Error | Section 6.6.1 |
| 501 | Not Implemented | Section 6.6.2 |
| 502 | Bad Gateway | Section 6.6.3 |
| 503 | Service Unavailable | Section 6.6.4 |
| 504 | Gateway Timeout | Section 6.6.5 |
| 505 | HTTP Version Not Supported | Section 6.6.6 |

8.3. Header Field Registry

HTTP header fields are registered within the Message Header Field Registry located at <http://www.iana.org/assignments/message-headers/message-header-index.html>, as defined by [BCP90].

8.3.1. Considerations for New Header Fields

Header fields are key:value pairs that can be used to communicate data about the message, its payload, the target resource, or the connection (i.e., control data). See Section 3.2 of [Part1] for a general definition of header field syntax in HTTP messages.

The requirements for header field names are defined in [BCP90].

Authors of specifications defining new fields are advised to keep the name as short as practical and to not prefix the name with "X-" unless the header field will never be used on the Internet. (The "x-" prefix idiom has been extensively misused in practice; it was intended to only be used as a mechanism for avoiding name collisions inside proprietary software or intranet processing, since the prefix would ensure that private names never collide with a newly registered Internet name; see [BCP178] for further information)

New header field values typically have their syntax defined using ABNF ([RFC5234]), using the extension defined in Section 7 of [Part1] as necessary, and are usually constrained to the range of ASCII characters. Header fields needing a greater range of characters can use an encoding such as the one defined in [RFC5987].

Leading and trailing whitespace in raw field values is removed upon field parsing (Section 3.2.4 of [Part1]). Field definitions where leading or trailing whitespace in values is significant will have to use a container syntax such as quoted-string (Section 3.2.6 of [Part1]).

Because commas (",") are used as a generic delimiter between field-values, they need to be treated with care if they are allowed in the field-value. Typically, components that might contain a comma are protected with double-quotes using the quoted-string ABNF production.

For example, a textual date and a URI (either of which might contain a comma) could be safely carried in field-values like these:

```
Example-URI-Field: "http://example.com/a.html,foo",  
                  "http://without-a-comma.example.com/"  
Example-Date-Field: "Sat, 04 May 1996", "Wed, 14 Sep 2005"
```

Note that double-quote delimiters almost always are used with the quoted-string production; using a different syntax inside double-quotes will likely cause unnecessary confusion.

Many header fields use a format including (case-insensitively) named parameters (for instance, Content-Type, defined in Section 3.1.1.5).

Allowing both unquoted (token) and quoted (quoted-string) syntax for the parameter value enables recipients to use existing parser components. When allowing both forms, the meaning of a parameter value ought to be independent of the syntax used for it (for an example, see the notes on parameter handling for media types in Section 3.1.1.1).

Authors of specifications defining new header fields are advised to consider documenting:

- o Whether the field is a single value, or whether it can be a list (delimited by commas; see Section 3.2 of [Part1]).

If it does not use the list syntax, document how to treat messages where the field occurs multiple times (a sensible default would be to ignore the field, but this might not always be the right choice).

Note that intermediaries and software libraries might combine multiple header field instances into a single one, despite the field's definition not allowing the list syntax. A robust format enables recipients to discover these situations (good example: "Content-Type", as the comma can only appear inside quoted strings; bad example: "Location", as a comma can occur inside a URI).

- o Under what conditions the header field can be used; e.g., only in responses or requests, in all messages, only on responses to a particular request method, etc.
- o Whether the field should be stored by origin servers that understand it upon a PUT request.
- o Whether the field semantics are further refined by the context, such as by existing request methods or status codes.
- o Whether it is appropriate to list the field-name in the Connection header field (i.e., if the header field is to be hop-by-hop; see Section 6.1 of [Part1]).
- o Under what conditions intermediaries are allowed to insert, delete, or modify the field's value.
- o Whether it is appropriate to list the field-name in a Vary response header field (e.g., when the request header field is used by an origin server's content selection algorithm; see Section 7.1.4).

- o Whether the header field is useful or allowable in trailers (see Section 4.1 of [Part1]).
- o Whether the header field ought to be preserved across redirects.
- o Whether it introduces any additional security considerations, such as disclosure of privacy-related data.

8.3.2. Registrations

The Message Header Field Registry shall be updated with the following permanent registrations:

| Header Field Name | Protocol | Status | Reference |
|-------------------|----------|----------|-----------------|
| Accept | http | standard | Section 5.3.2 |
| Accept-Charset | http | standard | Section 5.3.3 |
| Accept-Encoding | http | standard | Section 5.3.4 |
| Accept-Language | http | standard | Section 5.3.5 |
| Allow | http | standard | Section 7.4.1 |
| Content-Encoding | http | standard | Section 3.1.2.2 |
| Content-Language | http | standard | Section 3.1.3.2 |
| Content-Location | http | standard | Section 3.1.4.2 |
| Content-Type | http | standard | Section 3.1.1.5 |
| Date | http | standard | Section 7.1.1.2 |
| Expect | http | standard | Section 5.1.1 |
| From | http | standard | Section 5.5.1 |
| Location | http | standard | Section 7.1.2 |
| MIME-Version | http | standard | Appendix A.1 |
| Max-Forwards | http | standard | Section 5.1.2 |
| Referer | http | standard | Section 5.5.2 |
| Retry-After | http | standard | Section 7.1.3 |
| Server | http | standard | Section 7.4.2 |
| User-Agent | http | standard | Section 5.5.3 |
| Vary | http | standard | Section 7.1.4 |

The change controller for the above registrations is: "IETF (iesg@ietf.org) - Internet Engineering Task Force".

8.4. Content Coding Registry

The HTTP Content Coding Registry defines the name space for content coding names (Section 4.2 of [Part1]). The content coding registry is maintained at <http://www.iana.org/assignments/http-parameters>.

8.4.1. Procedure

Content Coding registrations MUST include the following fields:

- o Name
- o Description
- o Pointer to specification text

Names of content codings MUST NOT overlap with names of transfer codings (Section 4 of [Part1]), unless the encoding transformation is identical (as is the case for the compression codings defined in Section 4.2 of [Part1]).

Values to be added to this name space require IETF Review (see Section 4.1 of [RFC5226]), and MUST conform to the purpose of content coding defined in this section.

8.4.2. Registrations

The HTTP Content Codings Registry shall be updated with the registrations below:

| Name | Description | Reference |
|----------|---|---------------|
| identity | Reserved (synonym for "no encoding" in Accept-Encoding) | Section 5.3.4 |

9. Security Considerations

This section is meant to inform developers, information providers, and users of known security concerns relevant to HTTP semantics and its use for transferring information over the Internet. Considerations related to message syntax, parsing, and routing are discussed in Section 9 of [Part1].

The list of considerations below is not exhaustive. Most security concerns related to HTTP semantics are about securing server-side applications (code behind the HTTP interface), securing user agent processing of payloads received via HTTP, or secure use of the Internet in general, rather than security of the protocol. Various organizations maintain topical information and links to current research on Web application security (e.g., [OWASP]).

9.1. Attacks Based On File and Path Names

Origin servers frequently make use of their local file system to manage the mapping from effective request URI to resource representations. Implementers need to be aware that most file systems are not designed to protect against malicious file or path names, and thus depend on the origin server to avoid mapping to file names, folders, or directories that have special significance to the system.

For example, UNIX, Microsoft Windows, and other operating systems use ".." as a path component to indicate a directory level above the current one, and use specially named paths or file names to send data to system devices. Similar naming conventions might exist within other types of storage systems. Likewise, local storage systems have an annoying tendency to prefer user-friendliness over security when handling invalid or unexpected characters, recomposition of decomposed characters, and case-normalization of case-insensitive names.

Attacks based on such special names tend to focus on either denial of service (e.g., telling the server to read from a COM port) or disclosure of configuration and source files that are not meant to be served.

9.2. Attacks Based On Command, Code, or Query Injection

Origin servers often use parameters within the URI as a means of identifying system services, selecting database entries, or choosing a data source. However, data received in a request cannot be trusted. An attacker could construct any of the request data elements (method, request-target, header fields, or body) to contain data that might be misinterpreted as a command, code, or query when passed through a command invocation, language interpreter, or database interface.

For example, SQL injection is a common attack wherein additional query language is inserted within some part of the request-target or header fields (e.g., Host, Referer, etc.). If the received data is used directly within a SELECT statement, the query language might be interpreted as a database command instead of a simple string value. This type of implementation vulnerability is extremely common, in spite of being easy to prevent.

In general, resource implementations ought to avoid use of request data in contexts that are processed or interpreted as instructions. Parameters ought to be compared to fixed strings and acted upon as a result of that comparison, rather than passed through an interface

that is not prepared for untrusted data. Received data that isn't based on fixed parameters ought to be carefully filtered or encoded to avoid being misinterpreted.

Similar considerations apply to request data when it is stored and later processed, such as within log files, monitoring tools, or when included within a data format that allows embedded scripts.

9.3. Disclosure of Personal Information

Clients are often privy to large amounts of personal information, including both information provided by the user to interact with resources (e.g., the user's name, location, mail address, passwords, encryption keys, etc.) and information about the user's browsing activity over time (e.g., history, bookmarks, etc.). Implementations need to prevent unintentional disclosure of personal information.

9.4. Disclosure of Sensitive Information in URIs

URIs are intended to be shared, not secured, even when they identify secure resources. URIs are often shown on displays, added to templates when a page is printed, and stored in a variety of unprotected bookmark lists. It is therefore unwise to include information within a URI that is sensitive, personally identifiable, or a risk to disclose.

Authors of services ought to avoid GET-based forms for the submission of sensitive data because that data will be placed in the request-target. Many existing servers, proxies, and user agents log or display the request-target in places where it might be visible to third parties. Such services ought to use POST-based form submission instead.

Since the Referer header field tells a target site about the context that resulted in a request, it has the potential to reveal information about the user's immediate browsing history and any personal information that might be found in the referring resource's URI. Limitations on Referer are described in Section 5.5.2 to address some of its security considerations.

9.5. Disclosure of Fragment after Redirects

Although fragment identifiers used within URI references are not sent in requests, implementers ought to be aware that they will be visible to the user agent and any extensions or scripts running as a result of the response. In particular, when a redirect occurs and the original request's fragment identifier is inherited by the new reference in Location (Section 7.1.2), this might have the effect of

disclosing one site's fragment to another site. If the first site uses personal information in fragments, it ought to ensure that redirects to other sites include a (possibly empty) fragment component in order to block that inheritance.

9.6. Disclosure of Product Information

The User-Agent (Section 5.5.3), Via (Section 5.7.1 of [Part1]), and Server (Section 7.4.2) header fields often reveal information about the respective sender's software systems. In theory, this can make it easier for an attacker to exploit known security holes; in practice, attackers tend to try all potential holes regardless of the apparent software versions being used.

Proxies that serve as a portal through a network firewall ought to take special precautions regarding the transfer of header information that might identify hosts behind the firewall. The Via header field allows intermediaries to replace sensitive machine names with pseudonyms.

9.7. Browser Fingerprinting

Browser fingerprinting is a set of techniques for identifying a specific user agent over time through its unique set of characteristics. These characteristics might include information related to its TCP behavior, feature capabilities, and scripting environment, though of particular interest here is the set of unique characteristics that might be communicated via HTTP. Fingerprinting is considered a privacy concern because it enables tracking of a user agent's behavior over time without the corresponding controls that the user might have over other forms of data collection (e.g., cookies). Many general-purpose user agents (i.e., Web browsers) have taken steps to reduce their fingerprints.

There are a number of request header fields that might reveal information to servers that is sufficiently unique to enable fingerprinting. The From header field is the most obvious, though it is expected that From will only be sent when self-identification is desired by the user. Likewise, Cookie header fields are deliberately designed to enable re-identification, so fingerprinting concerns only apply to situations where cookies are disabled or restricted by the user agent's configuration.

The User-Agent header field might contain enough information to uniquely identify a specific device, usually when combined with other characteristics, particularly if the user agent sends excessive details about the user's system or extensions. However, the source of unique information that is least expected by users is proactive

negotiation (Section 5.3), including the Accept, Accept-Charset, Accept-Encoding, and Accept-Language header fields.

In addition to the fingerprinting concern, detailed use of the Accept-Language header field can reveal information the user might consider to be of a private nature. For example, understanding a given language set might be strongly correlated to membership in a particular ethnic group. An approach that limits such loss of privacy would be for a user agent to omit the sending of Accept-Language except for sites that have been whitelisted, perhaps via interaction after detecting a Vary header field that indicates language negotiation might be useful.

In environments where proxies are used to enhance privacy, user agents ought to be conservative in sending proactive negotiation header fields. General-purpose user agents that provide a high degree of header field configurability ought to inform users about the loss of privacy that might result if too much detail is provided. As an extreme privacy measure, proxies could filter the proactive negotiation header fields in relayed requests.

10. Acknowledgments

See Section 10 of [Part1].

11. References

11.1. Normative References

- [Part1] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", draft-ietf-httpbis-p1-messaging-26 (work in progress), February 2014.
- [Part4] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests", draft-ietf-httpbis-p4-conditional-26 (work in progress), February 2014.
- [Part5] Fielding, R., Ed., Lafon, Y., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Range Requests", draft-ietf-httpbis-p5-range-26 (work in progress), February 2014.
- [Part6] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Caching", draft-ietf-httpbis-p6-cache-26 (work in progress), February 2014.

- [Part7] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Authentication", draft-ietf-httpbis-p7-auth-26 (work in progress), February 2014.
- [RFC2045] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, November 1996.
- [RFC2046] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types", RFC 2046, November 1996.
- [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.
- [RFC4647] Phillips, A., Ed. and M. Davis, Ed., "Matching of Language Tags", BCP 47, RFC 4647, September 2006.
- [RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008.
- [RFC5646] Phillips, A., Ed. and M. Davis, Ed., "Tags for Identifying Languages", BCP 47, RFC 5646, September 2009.
- [RFC6365] Hoffman, P. and J. Klensin, "Terminology Used in Internationalization in the IETF", BCP 166, RFC 6365, September 2011.

11.2. Informative References

- [BCP13] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, January 2013.
- [BCP178] Saint-Andre, P., Crocker, D., and M. Nottingham, "Deprecating the "X-" Prefix and Similar Constructs in Application Protocols", BCP 178, RFC 6648, June 2012.
- [BCP90] Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, September 2004.

- [OWASP] van der Stock, A., Ed., "A Guide to Building Secure Web Applications and Web Services", The Open Web Application Security Project (OWASP) 2.0.1, July 2005, <<https://www.owasp.org/>>.
- [REST] Fielding, R., "Architectural Styles and the Design of Network-based Software Architectures", Doctoral Dissertation, University of California, Irvine, September 2000, <<http://roy.gbiv.com/pubs/dissertation/top.htm>>.
- [RFC1945] Berners-Lee, T., Fielding, R., and H. Nielsen, "Hypertext Transfer Protocol -- HTTP/1.0", RFC 1945, May 1996.
- [RFC2049] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part Five: Conformance Criteria and Examples", RFC 2049, November 1996.
- [RFC2068] Fielding, R., Gettys, J., Mogul, J., Nielsen, H., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2068, January 1997.
- [RFC2295] Holtman, K. and A. Mutz, "Transparent Content Negotiation in HTTP", RFC 2295, March 1998.
- [RFC2388] Masinter, L., "Returning Values from Forms: multipart/form-data", RFC 2388, August 1998.
- [RFC2557] Palme, F., Hopmann, A., Shelness, N., and E. Stefferud, "MIME Encapsulation of Aggregate Documents, such as HTML (MHTML)", RFC 2557, March 1999.
- [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.
- [RFC2774] Frystyk, H., Leach, P., and S. Lawrence, "An HTTP Extension Framework", RFC 2774, February 2000.
- [RFC2817] Khare, R. and S. Lawrence, "Upgrading to TLS Within HTTP/1.1", RFC 2817, May 2000.
- [RFC2978] Freed, N. and J. Postel, "IANA Charset Registration Procedures", BCP 19, RFC 2978, October 2000.
- [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.
- [RFC5322] Resnick, P., "Internet Message Format", RFC 5322, October 2008.
- [RFC5789] Dusseault, L. and J. Snell, "PATCH Method for HTTP", RFC 5789, March 2010.
- [RFC5905] Mills, D., Martin, J., Ed., Burbank, J., and W. Kasch, "Network Time Protocol Version 4: Protocol and Algorithms Specification", RFC 5905, June 2010.
- [RFC5987] Reschke, J., "Character Set and Language Encoding for Hypertext Transfer Protocol (HTTP) Header Field Parameters", RFC 5987, August 2010.
- [RFC5988] Nottingham, M., "Web Linking", RFC 5988, October 2010.
- [RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, April 2011.
- [RFC6266] Reschke, J., "Use of the Content-Disposition Header Field in the Hypertext Transfer Protocol (HTTP)", RFC 6266, June 2011.
- [status-308] Reschke, J., "The Hypertext Transfer Protocol (HTTP) Status Code 308 (Permanent Redirect)", draft-reschke-http-status-308-07 (work in progress), March 2012.

Appendix A. Differences between HTTP and MIME

HTTP/1.1 uses many of the constructs defined for the Internet Message Format [RFC5322] and the Multipurpose Internet Mail Extensions (MIME) [RFC2045] to allow a message body to be transmitted in an open variety of representations and with extensible header fields. However, RFC 2045 is focused only on email; applications of HTTP have many characteristics that differ from email, and hence HTTP has features that differ from MIME. These differences were carefully chosen to optimize performance over binary connections, to allow greater freedom in the use of new media types, to make date comparisons easier, and to acknowledge the practice of some early HTTP servers and clients.

This appendix describes specific areas where HTTP differs from MIME. Proxies and gateways to and from strict MIME environments need to be aware of these differences and provide the appropriate conversions where necessary.

A.1. MIME-Version

HTTP is not a MIME-compliant protocol. However, messages can include a single MIME-Version header field to indicate what version of the MIME protocol was used to construct the message. Use of the MIME-Version header field indicates that the message is in full conformance with the MIME protocol (as defined in [RFC2045]). Senders are responsible for ensuring full conformance (where possible) when exporting HTTP messages to strict MIME environments.

A.2. Conversion to Canonical Form

MIME requires that an Internet mail body part be converted to canonical form prior to being transferred, as described in Section 4 of [RFC2049]. Section 3.1.1.3 of this document describes the forms allowed for subtypes of the "text" media type when transmitted over HTTP. [RFC2046] requires that content with a type of "text" represent line breaks as CRLF and forbids the use of CR or LF outside of line break sequences. HTTP allows CRLF, bare CR, and bare LF to indicate a line break within text content.

A proxy or gateway from HTTP to a strict MIME environment ought to translate all line breaks within the text media types described in Section 3.1.1.3 of this document to the RFC 2049 canonical form of CRLF. Note, however, this might be complicated by the presence of a Content-Encoding and by the fact that HTTP allows the use of some charsets that do not use octets 13 and 10 to represent CR and LF, respectively.

Conversion will break any cryptographic checksums applied to the original content unless the original content is already in canonical form. Therefore, the canonical form is recommended for any content that uses such checksums in HTTP.

A.3. Conversion of Date Formats

HTTP/1.1 uses a restricted set of date formats (Section 7.1.1.1) to simplify the process of date comparison. Proxies and gateways from other protocols ought to ensure that any Date header field present in a message conforms to one of the HTTP/1.1 formats and rewrite the date if necessary.

A.4. Conversion of Content-Encoding

MIME does not include any concept equivalent to HTTP/1.1's Content-Encoding header field. Since this acts as a modifier on the media type, proxies and gateways from HTTP to MIME-compliant protocols ought to either change the value of the Content-Type header field or decode the representation before forwarding the message. (Some experimental applications of Content-Type for Internet mail have used a media-type parameter of ";conversions=<content-coding>" to perform a function equivalent to Content-Encoding. However, this parameter is not part of the MIME standards).

A.5. Conversion of Content-Transfer-Encoding

HTTP does not use the Content-Transfer-Encoding field of MIME. Proxies and gateways from MIME-compliant protocols to HTTP need to remove any Content-Transfer-Encoding prior to delivering the response message to an HTTP client.

Proxies and gateways from HTTP to MIME-compliant protocols are responsible for ensuring that the message is in the correct format and encoding for safe transport on that protocol, where "safe transport" is defined by the limitations of the protocol being used. Such a proxy or gateway ought to transform and label the data with an appropriate Content-Transfer-Encoding if doing so will improve the likelihood of safe transport over the destination protocol.

A.6. MHTML and Line Length Limitations

HTTP implementations that share code with MHTML [RFC2557] implementations need to be aware of MIME line length limitations. Since HTTP does not have this limitation, HTTP does not fold long lines. MHTML messages being transported by HTTP follow all conventions of MHTML, including line length limitations and folding, canonicalization, etc., since HTTP transfers message-bodies as payload and, aside from the "multipart/byteranges" type (Appendix A of [Part5]), does not interpret the content or any MIME header lines that might be contained therein.

Appendix B. Changes from RFC 2616

The primary changes in this revision have been editorial in nature: extracting the messaging syntax and partitioning HTTP semantics into separate documents for the core features, conditional requests, partial requests, caching, and authentication. The conformance language has been revised to clearly target requirements and the terminology has been improved to distinguish payload from representations and representations from resources.

A new requirement has been added that semantics embedded in a URI should be disabled when those semantics are inconsistent with the request method, since this is a common cause of interoperability failure. (Section 2)

An algorithm has been added for determining if a payload is associated with a specific identifier. (Section 3.1.4.1)

The default charset of ISO-8859-1 for text media types has been removed; the default is now whatever the media type definition says. Likewise, special treatment of ISO-8859-1 has been removed from the Accept-Charset header field. (Section 3.1.1.3 and Section 5.3.3)

The definition of Content-Location has been changed to no longer affect the base URI for resolving relative URI references, due to poor implementation support and the undesirable effect of potentially breaking relative links in content-negotiated resources. (Section 3.1.4.2)

To be consistent with the method-neutral parsing algorithm of [Part1], the definition of GET has been relaxed so that requests can have a body, even though a body has no meaning for GET. (Section 4.3.1)

Servers are no longer required to handle all Content-* header fields and use of Content-Range has been explicitly banned in PUT requests. (Section 4.3.4)

Definition of the CONNECT method has been moved from [RFC2817] to this specification. (Section 4.3.6)

The OPTIONS and TRACE request methods have been defined as being safe. (Section 4.3.7 and Section 4.3.8)

The Expect header field's extension mechanism has been removed due to widely-deployed broken implementations. (Section 5.1.1)

The Max-Forwards header field has been restricted to the OPTIONS and TRACE methods; previously, extension methods could have used it as well. (Section 5.1.2)

The "about:blank" URI has been suggested as a value for the Referer header field when no referring URI is applicable, which distinguishes that case from others where the Referer field is not sent or has been removed. (Section 5.5.2)

The following status codes are now cacheable (that is, they can be stored and reused by a cache without explicit freshness information

present): 204, 404, 405, 414, 501. (Section 6)

The 201 (Created) status description has been changed to allow for the possibility that more than one resource has been created. (Section 6.3.2)

The definition of 203 (Non-Authoritative Information) has been broadened to include cases of payload transformations as well. (Section 6.3.4)

The set of request methods that are safe to automatically redirect is no longer closed; user agents are able to make that determination based upon the request method semantics. The redirect status codes 301, 302, and 307 no longer have normative requirements on response payloads and user interaction. (Section 6.4)

The status codes 301 and 302 have been changed to allow user agents to rewrite the method from POST to GET. (Sections 6.4.2 and 6.4.3)

The description of 303 (See Other) status code has been changed to allow it to be cached if explicit freshness information is given, and a specific definition has been added for a 303 response to GET. (Section 6.4.4)

The 305 (Use Proxy) status code has been deprecated due to security concerns regarding in-band configuration of a proxy. (Section 6.4.5)

The 400 (Bad Request) status code has been relaxed so that it isn't limited to syntax errors. (Section 6.5.1)

The 426 (Upgrade Required) status code has been incorporated from [RFC2817]. (Section 6.5.15)

The target of requirements on HTTP-date and the Date header field have been reduced to those systems generating the date, rather than all systems sending a date. (Section 7.1.1)

The syntax of the Location header field has been changed to allow all URI references, including relative references and fragments, along with some clarifications as to when use of fragments would not be appropriate. (Section 7.1.2)

Allow has been reclassified as a response header field, removing the option to specify it in a PUT request. Requirements relating to the content of Allow have been relaxed; correspondingly, clients are not required to always trust its value. (Section 7.4.1)

A Method Registry has been defined. (Section 8.1)

The Status Code Registry has been redefined by this specification; previously, it was defined in Section 7.1 of [RFC2817]. (Section 8.2)

Registration of Content Codings has been changed to require IETF Review. (Section 8.4)

The Content-Disposition header field has been removed since it is now defined by [RFC6266].

The Content-MD5 header field has been removed because it was inconsistently implemented with respect to partial responses.

Appendix C. Imported ABNF

The following core rules are included by reference, as defined in Appendix B.1 of [RFC5234]: ALPHA (letters), CR (carriage return), CRLF (CR LF), CTL (controls), DIGIT (decimal 0-9), DQUOTE (double quote), HEXDIG (hexadecimal 0-9/A-F/a-f), HTAB (horizontal tab), LF (line feed), OCTET (any 8-bit sequence of data), SP (space), and VCHAR (any visible US-ASCII character).

The rules below are defined in [Part1]:

```
BWS           = <BWS, defined in [Part1], Section 3.2.3>
OWS           = <OWS, defined in [Part1], Section 3.2.3>
RWS           = <RWS, defined in [Part1], Section 3.2.3>
URI-reference = <URI-reference, defined in [Part1], Section 2.7>
absolute-URI  = <absolute-URI, defined in [Part1], Section 2.7>
comment       = <comment, defined in [Part1], Section 3.2.6>
field-name    = <comment, defined in [Part1], Section 3.2>
partial-URI   = <partial-URI, defined in [Part1], Section 2.7>
quoted-string = <quoted-string, defined in [Part1], Section 3.2.6>
token         = <token, defined in [Part1], Section 3.2.6>
```

Appendix D. Collected ABNF

In the collected ABNF below, list rules are expanded as per Section 1.2 of [Part1].

```
Accept = [ ( "," / ( media-range [ accept-params ] ) ) *( OWS "," [
  OWS ( media-range [ accept-params ] ) ] ) ]
Accept-Charset = *( "," OWS ) ( ( charset / "*" ) [ weight ] ) *( OWS
  "," [ OWS ( ( charset / "*" ) [ weight ] ) ] )
Accept-Encoding = [ ( "," / ( codings [ weight ] ) ) *( OWS "," [ OWS
  ( codings [ weight ] ) ] ) ]
Accept-Language = *( "," OWS ) ( language-range [ weight ] ) *( OWS
  "," [ OWS ( language-range [ weight ] ) ] )
```

```
Allow = [ ( "," / method ) *( OWS "," [ OWS method ] ) ]

BWS = <BWS, defined in [Part1], Section 3.2.3>

Content-Encoding = *( "," OWS ) content-coding *( OWS "," [ OWS
  content-coding ] )
Content-Language = *( "," OWS ) language-tag *( OWS "," [ OWS
  language-tag ] )
Content-Location = absolute-URI / partial-URI
Content-Type = media-type

Date = HTTP-date

Expect = "100-continue"

From = mailbox

GMT = %x47.4D.54 ; GMT

HTTP-date = IMF-fixdate / obs-date

IMF-fixdate = day-name "," SP date1 SP time-of-day SP GMT

Location = URI-reference

Max-Forwards = 1*DIGIT

OWS = <OWS, defined in [Part1], Section 3.2.3>

RWS = <RWS, defined in [Part1], Section 3.2.3>
Referer = absolute-URI / partial-URI
Retry-After = HTTP-date / delay-seconds

Server = product *( RWS ( product / comment ) )

URI-reference = <URI-reference, defined in [Part1], Section 2.7>
User-Agent = product *( RWS ( product / comment ) )

Vary = "*" / ( *( "," OWS ) field-name *( OWS "," [ OWS field-name ]
  ) )

absolute-URI = <absolute-URI, defined in [Part1], Section 2.7>
accept-ext = OWS ";" OWS token [ "=" ( token / quoted-string ) ]
accept-params = weight *accept-ext
asctime-date = day-name SP date3 SP time-of-day SP year

charset = token
codings = content-coding / "identity" / ""
```

```
comment = <comment, defined in [Part1], Section 3.2.6>
content-coding = token

date1 = day SP month SP year
date2 = day "-" month "-" 2DIGIT
date3 = month SP ( 2DIGIT / ( SP DIGIT ) )
day = 2DIGIT
day-name = %x4D.6F.6E ; Mon
           / %x54.75.65 ; Tue
           / %x57.65.64 ; Wed
           / %x54.68.75 ; Thu
           / %x46.72.69 ; Fri
           / %x53.61.74 ; Sat
           / %x53.75.6E ; Sun
day-name-l = %x4D.6F.6E.64.61.79 ; Monday
             / %x54.75.65.73.64.61.79 ; Tuesday
             / %x57.65.64.6E.65.73.64.61.79 ; Wednesday
             / %x54.68.75.72.73.64.61.79 ; Thursday
             / %x46.72.69.64.61.79 ; Friday
             / %x53.61.74.75.72.64.61.79 ; Saturday
             / %x53.75.6E.64.61.79 ; Sunday
delay-seconds = 1*DIGIT

field-name = <comment, defined in [Part1], Section 3.2>

hour = 2DIGIT

language-range = <language-range, defined in [RFC4647], Section 2.1>
language-tag = <Language-Tag, defined in [RFC5646], Section 2.1>

mailbox = <mailbox, defined in [RFC5322], Section 3.4>
media-range = ( "*" / ( type "/" ) / ( type "/" subtype ) ) *( OWS
    ";" OWS parameter )
media-type = type "/" subtype *( OWS ";" OWS parameter )
method = token
minute = 2DIGIT
month = %x4A.61.6E ; Jan
        / %x46.65.62 ; Feb
        / %x4D.61.72 ; Mar
        / %x41.70.72 ; Apr
        / %x4D.61.79 ; May
        / %x4A.75.6E ; Jun
        / %x4A.75.6C ; Jul
        / %x41.75.67 ; Aug
        / %x53.65.70 ; Sep
        / %x4F.63.74 ; Oct
        / %x4E.6F.76 ; Nov
        / %x44.65.63 ; Dec
```

```
obs-date = rfc850-date / asctime-date

parameter = token "=" ( token / quoted-string )
partial-URI = <partial-URI, defined in [Part1], Section 2.7>
product = token [ "/" product-version ]
product-version = token

quoted-string = <quoted-string, defined in [Part1], Section 3.2.6>
qvalue = ( "0" [ "." *3DIGIT ] ) / ( "1" [ "." *3"0" ] )

rfc850-date = day-name-1 "," SP date2 SP time-of-day SP GMT

second = 2DIGIT
subtype = token

time-of-day = hour ":" minute ":" second
token = <token, defined in [Part1], Section 3.2.6>
type = token

weight = OWS ";" OWS "q=" qvalue

year = 4DIGIT
```

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

E.1. Since RFC 2616

Changes up to the IETF Last Call draft are summarized in <<http://trac.tools.ietf.org/html/draft-ietf-httpbis-p2-semantics-24#appendix-E>>.

E.2. Since draft-ietf-httpbis-p2-semantics-24

Closed issues:

- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/432>>: "Review Cachability of Status Codes WRT 'Negative Caching'"
- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/499>>: "RFC 1305 ref needs to be updated to 5905"
- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/501>>: "idempotency: clarify 'effect'"
- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/503>>: "APPSDIR review of draft-ietf-httpbis-p2-semantics-24"

- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/517>>: "move IANA registrations to correct draft"

E.3. Since draft-ietf-httpbis-p2-semantics-25

Closed issues:

- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/520>>: "Gen-Art review of draft-ietf-httpbis-p2-semantics-24 with security considerations"
- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/532>>: "IESG ballot on draft-ietf-httpbis-p2-semantics-25"
- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/538>>: "add 'stateless' to Abstract"
- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/542>>: "improve introduction of list rule"
- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/545>>: "requirement on implementing methods according to their semantics"
- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/546>>: "considerations for new headers: privacy"
- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/549>>: "augment security considerations with pointers to current research"

Index

| | | |
|---|---------------------------------------|----|
| 1 | 1xx Informational (status code class) | 49 |
| 2 | 2xx Successful (status code class) | 50 |
| 3 | 3xx Redirection (status code class) | 53 |
| 4 | 4xx Client Error (status code class) | 57 |
| 5 | 5xx Server Error (status code class) | 61 |
| 1 | 100 Continue (status code) | 49 |

| | | |
|---|---|----|
| | 100-continue (expect value) | 33 |
| | 101 Switching Protocols (status code) | 49 |
| 2 | | |
| | 200 OK (status code) | 50 |
| | 201 Created (status code) | 51 |
| | 202 Accepted (status code) | 51 |
| | 203 Non-Authoritative Information (status code) | 51 |
| | 204 No Content (status code) | 52 |
| | 205 Reset Content (status code) | 52 |
| 3 | | |
| | 300 Multiple Choices (status code) | 54 |
| | 301 Moved Permanently (status code) | 55 |
| | 302 Found (status code) | 55 |
| | 303 See Other (status code) | 56 |
| | 305 Use Proxy (status code) | 56 |
| | 306 (Unused) (status code) | 56 |
| | 307 Temporary Redirect (status code) | 57 |
| 4 | | |
| | 400 Bad Request (status code) | 57 |
| | 402 Payment Required (status code) | 57 |
| | 403 Forbidden (status code) | 57 |
| | 404 Not Found (status code) | 58 |
| | 405 Method Not Allowed (status code) | 58 |
| | 406 Not Acceptable (status code) | 58 |
| | 408 Request Timeout (status code) | 59 |
| | 409 Conflict (status code) | 59 |
| | 410 Gone (status code) | 59 |
| | 411 Length Required (status code) | 60 |
| | 413 Payload Too Large (status code) | 60 |
| | 414 URI Too Long (status code) | 60 |
| | 415 Unsupported Media Type (status code) | 60 |
| | 417 Expectation Failed (status code) | 61 |
| | 426 Upgrade Required (status code) | 61 |
| 5 | | |
| | 500 Internal Server Error (status code) | 61 |
| | 501 Not Implemented (status code) | 62 |
| | 502 Bad Gateway (status code) | 62 |
| | 503 Service Unavailable (status code) | 62 |
| | 504 Gateway Timeout (status code) | 62 |
| | 505 HTTP Version Not Supported (status code) | 62 |
| A | | |
| | Accept header field | 38 |
| | Accept-Charset header field | 40 |

Accept-Encoding header field 41
Accept-Language header field 42
Allow header field 71

C

cacheable 24
compress (content coding) 11
conditional request 36
CONNECT method 30
content coding 11
content negotiation 6
Content-Encoding header field 12
Content-Language header field 13
Content-Location header field 15
Content-Transfer-Encoding header field 89
Content-Type header field 10

D

Date header field 66
deflate (content coding) 11
DELETE method 29

E

Expect header field 33

F

From header field 44

G

GET method 24
Grammar
 Accept 38
 Accept-Charset 40
 Accept-Encoding 41
 accept-ext 38
 Accept-Language 42
 accept-params 38
 Allow 71
 asctime-date 66
 charset 9
 codings 41
 content-coding 11
 Content-Encoding 12
 Content-Language 13
 Content-Location 15
 Content-Type 10
 Date 66
 date1 65

day 65
day-name 65
day-name-l 65
delay-seconds 69
Expect 34
From 44
GMT 65
hour 65
HTTP-date 63
IMF-fixdate 65
language-range 42
language-tag 13
Location 67
Max-Forwards 36
media-range 38
media-type 8
method 21
minute 65
month 65
obs-date 65
parameter 8
product 46
product-version 46
qvalue 38
Referer 45
Retry-After 69
rfc850-date 66
second 65
Server 72
subtype 8
time-of-day 65
type 8
User-Agent 46
Vary 69
weight 38
year 65
gzip (content coding) 11

H

HEAD method 25

I

idempotent 23

L

Location header field 67

M

| | | |
|---|-----------------------------|-------|
| | Max-Forwards header field | 36 |
| | MIME-Version header field | 88 |
| O | | |
| | OPTIONS method | 31 |
| P | | |
| | payload | 17 |
| | POST method | 25 |
| | PUT method | 26 |
| R | | |
| | Referer header field | 44 |
| | representation | 7 |
| | Retry-After header field | 68 |
| S | | |
| | safe | 22 |
| | selected representation | 7, 70 |
| | Server header field | 72 |
| | Status Codes Classes | |
| | 1xx Informational | 49 |
| | 2xx Successful | 50 |
| | 3xx Redirection | 53 |
| | 4xx Client Error | 57 |
| | 5xx Server Error | 61 |
| T | | |
| | TRACE method | 32 |
| U | | |
| | User-Agent header field | 46 |
| V | | |
| | Vary header field | 69 |
| X | | |
| | x-compress (content coding) | 11 |
| | x-gzip (content coding) | 11 |

Authors' Addresses

Roy T. Fielding (editor)
Adobe Systems Incorporated
345 Park Ave
San Jose, CA 95110
USA

EMail: fielding@gbiv.com
URI: <http://roy.gbiv.com/>

Julian F. Reschke (editor)
greenbytes GmbH
Hafenweg 16
Muenster, NW 48155
Germany

EMail: julian.reschke@greenbytes.de
URI: <http://greenbytes.de/tech/webdav/>

HTTPbis Working Group
Internet-Draft
Obsoletes: 2616 (if approved)
Intended status: Standards Track
Expires: August 10, 2014

R. Fielding, Ed.
Adobe
J. Reschke, Ed.
greenbytes
February 6, 2014

Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests
draft-ietf-httpbis-p4-conditional-26

Abstract

The Hypertext Transfer Protocol (HTTP) is a stateless application-level protocol for distributed, collaborative, hypertext information systems. This document defines HTTP/1.1 conditional requests, including metadata header fields for indicating state changes, request header fields for making preconditions on such state, and rules for constructing the responses to a conditional request when one or more preconditions evaluate to false.

Editorial Note (To be removed by RFC Editor)

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

The current issues list is at <http://tools.ietf.org/wg/httpbis/trac/report/3> and related documents (including fancy diffs) can be found at <http://tools.ietf.org/wg/httpbis/>.

The changes in this draft are summarized in Appendix 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 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.

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.

Table of Contents

| | |
|--|----|
| 1. Introduction | 4 |
| 1.1. Conformance and Error Handling | 4 |
| 1.2. Syntax Notation | 4 |
| 2. Validators | 5 |
| 2.1. Weak versus Strong | 5 |
| 2.2. Last-Modified | 7 |
| 2.2.1. Generation | 7 |
| 2.2.2. Comparison | 8 |
| 2.3. ETag | 9 |
| 2.3.1. Generation | 10 |
| 2.3.2. Comparison | 10 |
| 2.3.3. Example: Entity-tags Varying on Content-Negotiated Resources | 11 |
| 2.4. When to Use Entity-tags and Last-Modified Dates | 12 |
| 3. Precondition Header Fields | 13 |
| 3.1. If-Match | 13 |
| 3.2. If-None-Match | 14 |
| 3.3. If-Modified-Since | 15 |
| 3.4. If-Unmodified-Since | 16 |
| 3.5. If-Range | 18 |
| 4. Status Code Definitions | 18 |
| 4.1. 304 Not Modified | 18 |
| 4.2. 412 Precondition Failed | 18 |
| 5. Evaluation | 19 |
| 6. Precedence | 19 |
| 7. IANA Considerations | 21 |
| 7.1. Status Code Registration | 21 |
| 7.2. Header Field Registration | 21 |
| 8. Security Considerations | 22 |
| 9. Acknowledgments | 22 |
| 10. References | 23 |
| 10.1. Normative References | 23 |
| 10.2. Informative References | 23 |
| Appendix A. Changes from RFC 2616 | 23 |
| Appendix B. Imported ABNF | 24 |
| Appendix C. Collected ABNF | 24 |
| Appendix D. Change Log (to be removed by RFC Editor before publication) | 25 |
| D.1. Since draft-ietf-httpbis-p4-conditional-24 | 25 |
| D.2. Since draft-ietf-httpbis-p4-conditional-25 | 25 |
| Index | 26 |

1. Introduction

Conditional requests are HTTP requests [Part2] that include one or more header fields indicating a precondition to be tested before applying the method semantics to the target resource. This document defines the HTTP/1.1 conditional request mechanisms in terms of the architecture, syntax notation, and conformance criteria defined in [Part1].

Conditional GET requests are the most efficient mechanism for HTTP cache updates [Part6]. Conditionals can also be applied to state-changing methods, such as PUT and DELETE, to prevent the "lost update" problem: one client accidentally overwriting the work of another client that has been acting in parallel.

Conditional request preconditions are based on the state of the target resource as a whole (its current value set) or the state as observed in a previously obtained representation (one value in that set). A resource might have multiple current representations, each with its own observable state. The conditional request mechanisms assume that the mapping of requests to a "selected representation" (Section 3 of [Part2]) will be consistent over time if the server intends to take advantage of conditionals. Regardless, if the mapping is inconsistent and the server is unable to select the appropriate representation, then no harm will result when the precondition evaluates to false.

The conditional request preconditions defined by this specification (Section 3) are evaluated when applicable to the recipient (Section 5) according to their order of precedence (Section 6).

1.1. Conformance and Error Handling

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].

Conformance criteria and considerations regarding error handling are defined in Section 2.5 of [Part1].

1.2. Syntax Notation

This specification uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234] with a list extension, defined in Section 7 of [Part1], that allows for compact definition of comma-separated lists using a '#' operator (similar to how the '*' operator indicates repetition). Appendix B describes rules imported from other documents. Appendix C shows the collected grammar with all list

operators expanded to standard ABNF notation.

2. Validators

This specification defines two forms of metadata that are commonly used to observe resource state and test for preconditions: modification dates (Section 2.2) and opaque entity tags (Section 2.3). Additional metadata that reflects resource state has been defined by various extensions of HTTP, such as WebDAV [RFC4918], that are beyond the scope of this specification. A resource metadata value is referred to as a "validator" when it is used within a precondition.

2.1. Weak versus Strong

Validators come in two flavors: strong or weak. Weak validators are easy to generate but are far less useful for comparisons. Strong validators are ideal for comparisons but can be very difficult (and occasionally impossible) to generate efficiently. Rather than impose that all forms of resource adhere to the same strength of validator, HTTP exposes the type of validator in use and imposes restrictions on when weak validators can be used as preconditions.

A "strong validator" is representation metadata that changes value whenever a change occurs to the representation data that would be observable in the payload body of a 200 (OK) response to GET.

A strong validator might change for reasons other than a change to the representation data, such as when a semantically significant part of the representation metadata is changed (e.g., Content-Type), but it is in the best interests of the origin server to only change the value when it is necessary to invalidate the stored responses held by remote caches and authoring tools.

Cache entries might persist for arbitrarily long periods, regardless of expiration times. Thus, a cache might attempt to validate an entry using a validator that it obtained in the distant past. A strong validator is unique across all versions of all representations associated with a particular resource over time. However, there is no implication of uniqueness across representations of different resources (i.e., the same strong validator might be in use for representations of multiple resources at the same time and does not imply that those representations are equivalent).

There are a variety of strong validators used in practice. The best are based on strict revision control, wherein each change to a representation always results in a unique node name and revision identifier being assigned before the representation is made

accessible to GET. A collision-resistant hash function applied to the representation data is also sufficient if the data is available prior to the response header fields being sent and the digest does not need to be recalculated every time a validation request is received. However, if a resource has distinct representations that differ only in their metadata, such as might occur with content negotiation over media types that happen to share the same data format, then the origin server needs to incorporate additional information in the validator to distinguish those representations.

In contrast, a "weak validator" is representation metadata that might not change for every change to the representation data. This weakness might be due to limitations in how the value is calculated, such as clock resolution or an inability to ensure uniqueness for all possible representations of the resource, or due to a desire by the resource owner to group representations by some self-determined set of equivalency rather than unique sequences of data. An origin server **SHOULD** change a weak entity-tag whenever it considers prior representations to be unacceptable as a substitute for the current representation. In other words, a weak entity-tag ought to change whenever the origin server wants caches to invalidate old responses.

For example, the representation of a weather report that changes in content every second, based on dynamic measurements, might be grouped into sets of equivalent representations (from the origin server's perspective) with the same weak validator in order to allow cached representations to be valid for a reasonable period of time (perhaps adjusted dynamically based on server load or weather quality). Likewise, a representation's modification time, if defined with only one-second resolution, might be a weak validator if it is possible for the representation to be modified twice during a single second and retrieved between those modifications.

Likewise, a validator is weak if it is shared by two or more representations of a given resource at the same time, unless those representations have identical representation data. For example, if the origin server sends the same validator for a representation with a gzip content coding applied as it does for a representation with no content coding, then that validator is weak. However, two simultaneous representations might share the same strong validator if they differ only in the representation metadata, such as when two different media types are available for the same representation data.

Strong validators are usable for all conditional requests, including cache validation, partial content ranges, and "lost update" avoidance. Weak validators are only usable when the client does not require exact equality with previously obtained representation data, such as when validating a cache entry or limiting a web traversal to

recent changes.

2.2. Last-Modified

The "Last-Modified" header field in a response provides a timestamp indicating the date and time at which the origin server believes the selected representation was last modified, as determined at the conclusion of handling the request.

Last-Modified = HTTP-date

An example of its use is

Last-Modified: Tue, 15 Nov 1994 12:45:26 GMT

2.2.1. Generation

An origin server SHOULD send Last-Modified for any selected representation for which a last modification date can be reasonably and consistently determined, since its use in conditional requests and evaluating cache freshness ([Part6]) results in a substantial reduction of HTTP traffic on the Internet and can be a significant factor in improving service scalability and reliability.

A representation is typically the sum of many parts behind the resource interface. The last-modified time would usually be the most recent time that any of those parts were changed. How that value is determined for any given resource is an implementation detail beyond the scope of this specification. What matters to HTTP is how recipients of the Last-Modified header field can use its value to make conditional requests and test the validity of locally cached responses.

An origin server SHOULD obtain the Last-Modified value of the representation as close as possible to the time that it generates the Date field value for its response. This allows a recipient to make an accurate assessment of the representation's modification time, especially if the representation changes near the time that the response is generated.

An origin server with a clock MUST NOT send a Last-Modified date that is later than the server's time of message origination (Date). If the last modification time is derived from implementation-specific metadata that evaluates to some time in the future, according to the origin server's clock, then the origin server MUST replace that value with the message origination date. This prevents a future modification date from having an adverse impact on cache validation.

An origin server without a clock **MUST NOT** assign Last-Modified values to a response unless these values were associated with the resource by some other system or user with a reliable clock.

2.2.2. Comparison

A Last-Modified time, when used as a validator in a request, is implicitly weak unless it is possible to deduce that it is strong, using the following rules:

- o The validator is being compared by an origin server to the actual current validator for the representation and,
- o That origin server reliably knows that the associated representation did not change twice during the second covered by the presented validator.

or

- o The validator is about to be used by a client in an If-Modified-Since, If-Unmodified-Since header field, because the client has a cache entry, or If-Range for the associated representation, and
- o That cache entry includes a Date value, which gives the time when the origin server sent the original response, and
- o The presented Last-Modified time is at least 60 seconds before the Date value.

or

- o The validator is being compared by an intermediate cache to the validator stored in its cache entry for the representation, and
- o That cache entry includes a Date value, which gives the time when the origin server sent the original response, and
- o The presented Last-Modified time is at least 60 seconds before the Date value.

This method relies on the fact that if two different responses were sent by the origin server during the same second, but both had the same Last-Modified time, then at least one of those responses would have a Date value equal to its Last-Modified time. The arbitrary 60-second limit guards against the possibility that the Date and Last-Modified values are generated from different clocks, or at somewhat different times during the preparation of the response. An implementation **MAY** use a value larger than 60 seconds, if it is

believed that 60 seconds is too short.

2.3. ETag

The "ETag" header field in a response provides the current entity-tag for the selected representation, as determined at the conclusion of handling the request. An entity-tag is an opaque validator for differentiating between multiple representations of the same resource, regardless of whether those multiple representations are due to resource state changes over time, content negotiation resulting in multiple representations being valid at the same time, or both. An entity-tag consists of an opaque quoted string, possibly prefixed by a weakness indicator.

```
ETag          = entity-tag

entity-tag    = [ weak ] opaque-tag
weak          = %x57.2F ; "W/", case-sensitive
opaque-tag    = DQUOTE *etagc DQUOTE
etagc         = %x21 / %x23-7E / obs-text
               ; VCHAR except double quotes, plus obs-text
```

Note: Previously, opaque-tag was defined to be a quoted-string ([RFC2616], Section 3.11), thus some recipients might perform backslash unescaping. Servers therefore ought to avoid backslash characters in entity tags.

An entity-tag can be more reliable for validation than a modification date in situations where it is inconvenient to store modification dates, where the one-second resolution of HTTP date values is not sufficient, or where modification dates are not consistently maintained.

Examples:

```
ETag: "xyzzy"
ETag: W/"xyzzy"
ETag: ""
```

An entity-tag can be either a weak or strong validator, with strong being the default. If an origin server provides an entity-tag for a representation and the generation of that entity-tag does not satisfy all of the characteristics of a strong validator (Section 2.1), then the origin server MUST mark the entity-tag as weak by prefixing its opaque value with "W/" (case-sensitive).

2.3.1. Generation

The principle behind entity-tags is that only the service author knows the implementation of a resource well enough to select the most accurate and efficient validation mechanism for that resource, and that any such mechanism can be mapped to a simple sequence of octets for easy comparison. Since the value is opaque, there is no need for the client to be aware of how each entity-tag is constructed.

For example, a resource that has implementation-specific versioning applied to all changes might use an internal revision number, perhaps combined with a variance identifier for content negotiation, to accurately differentiate between representations. Other implementations might use a collision-resistant hash of representation content, a combination of various file attributes, or a modification timestamp that has sub-second resolution.

An origin server **SHOULD** send ETag for any selected representation for which detection of changes can be reasonably and consistently determined, since the entity-tag's use in conditional requests and evaluating cache freshness ([Part6]) can result in a substantial reduction of HTTP network traffic and can be a significant factor in improving service scalability and reliability.

2.3.2. Comparison

There are two entity-tag comparison functions, depending on whether the comparison context allows the use of weak validators or not:

- o Strong comparison: two entity-tags are equivalent if both are not weak and their opaque-tags match character-by-character.
- o Weak comparison: two entity-tags are equivalent if their opaque-tags match character-by-character, regardless of either or both being tagged as "weak".

The example below shows the results for a set of entity-tag pairs, and both the weak and strong comparison function results:

| ETag 1 | ETag 2 | Strong Comparison | Weak Comparison |
|--------|--------|-------------------|-----------------|
| W/"1" | W/"1" | no match | match |
| W/"1" | W/"2" | no match | no match |
| W/"1" | "1" | no match | match |
| "1" | "1" | match | match |

2.3.3. Example: Entity-tags Varying on Content-Negotiated Resources

Consider a resource that is subject to content negotiation (Section 3.4 of [Part2]), and where the representations sent in response to a GET request vary based on the Accept-Encoding request header field (Section 5.3.4 of [Part2]):

>> Request:

```
GET /index HTTP/1.1
Host: www.example.com
Accept-Encoding: gzip
```

In this case, the response might or might not use the gzip content coding. If it does not, the response might look like:

>> Response:

```
HTTP/1.1 200 OK
Date: Fri, 26 Mar 2010 00:05:00 GMT
ETag: "123-a"
Content-Length: 70
Vary: Accept-Encoding
Content-Type: text/plain

Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

An alternative representation that does use gzip content coding would be:

>> Response:

```
HTTP/1.1 200 OK
Date: Fri, 26 Mar 2010 00:05:00 GMT
ETag: "123-b"
Content-Length: 43
Vary: Accept-Encoding
Content-Type: text/plain
Content-Encoding: gzip

...binary data...
```

Note: Content codings are a property of the representation data, so a strong entity-tag for a content-encoded representation has to be distinct from the entity tag of an unencoded representation to prevent potential conflicts during cache updates and range requests. In contrast, transfer codings (Section 4 of [Part1]) apply only during message transfer and do not result in distinct entity-tags.

2.4. When to Use Entity-tags and Last-Modified Dates

In 200 (OK) responses to GET or HEAD, an origin server:

- o SHOULD send an entity-tag validator unless it is not feasible to generate one.
- o MAY send a weak entity-tag instead of a strong entity-tag, if performance considerations support the use of weak entity-tags, or if it is unfeasible to send a strong entity-tag.
- o SHOULD send a Last-Modified value if it is feasible to send one.

In other words, the preferred behavior for an origin server is to send both a strong entity-tag and a Last-Modified value in successful responses to a retrieval request.

A client:

- o MUST send that entity-tag in any cache validation request (using If-Match or If-None-Match) if an entity-tag has been provided by the origin server.
- o SHOULD send the Last-Modified value in non-subrange cache validation requests (using If-Modified-Since) if only a Last-Modified value has been provided by the origin server.
- o MAY send the Last-Modified value in subrange cache validation requests (using If-Unmodified-Since) if only a Last-Modified value has been provided by an HTTP/1.0 origin server. The user agent SHOULD provide a way to disable this, in case of difficulty.
- o SHOULD send both validators in cache validation requests if both an entity-tag and a Last-Modified value have been provided by the origin server. This allows both HTTP/1.0 and HTTP/1.1 caches to respond appropriately.

3. Precondition Header Fields

This section defines the syntax and semantics of HTTP/1.1 header fields for applying preconditions on requests. Section 5 defines when the preconditions are applied. Section 6 defines the order of evaluation when more than one precondition is present.

3.1. If-Match

The "If-Match" header field makes the request method conditional on the recipient origin server either having at least one current representation of the target resource, when the field-value is "*", or having a current representation of the target resource that has an entity-tag matching a member of the list of entity-tags provided in the field-value.

An origin server **MUST** use the strong comparison function when comparing entity-tags for If-Match (Section 2.3.2), since the client intends this precondition to prevent the method from being applied if there have been any changes to the representation data.

If-Match = "*" / 1#entity-tag

Examples:

```
If-Match: "xyzzy"
If-Match: "xyzzy", "r2d2xxxx", "c3piozzzz"
If-Match: *
```

If-Match is most often used with state-changing methods (e.g., POST, PUT, DELETE) to prevent accidental overwrites when multiple user agents might be acting in parallel on the same resource (i.e., to prevent the "lost update" problem). It can also be used with safe methods to abort a request if the selected representation does not match one already stored (or partially stored) from a prior request.

An origin server that receives an If-Match header field **MUST** evaluate the condition prior to performing the method (Section 5). If the field-value is "*", the condition is false if the origin server does not have a current representation for the target resource. If the field-value is a list of entity-tags, the condition is false if none of the listed tags match the entity-tag of the selected representation.

An origin server **MUST NOT** perform the requested method if a received If-Match condition evaluates to false; instead the origin server **MUST** respond with either: a) the 412 (Precondition Failed) status code; or, b) one of the 2xx (Successful) status codes if the origin server

has verified that a state change is being requested and the final state is already reflected in the current state of the target resource (i.e., the change requested by the user agent has already succeeded, but the user agent might not be aware of it, perhaps because the prior response was lost or a compatible change was made by some other user agent). In the latter case, the origin server **MUST NOT** send a validator header field in the response unless it can verify that the request is a duplicate of an immediately prior change made by the same user agent.

The If-Match header field can be ignored by caches and intermediaries because it is not applicable to a stored response.

3.2. If-None-Match

The "If-None-Match" header field makes the request method conditional on a recipient cache or origin server either not having any current representation of the target resource, when the field-value is "*", or having a selected representation with an entity-tag that does not match any of those listed in the field-value.

A recipient **MUST** use the weak comparison function when comparing entity-tags for If-None-Match (Section 2.3.2), since weak entity-tags can be used for cache validation even if there have been changes to the representation data.

If-None-Match = "*" / 1#entity-tag

Examples:

```
If-None-Match: "xyzzy"
If-None-Match: W/"xyzzy"
If-None-Match: "xyzzy", "r2d2xxxx", "c3piozzzz"
If-None-Match: W/"xyzzy", W/"r2d2xxxx", W/"c3piozzzz"
If-None-Match: *
```

If-None-Match is primarily used in conditional GET requests to enable efficient updates of cached information with a minimum amount of transaction overhead. When a client desires to update one or more stored responses that have entity-tags, the client **SHOULD** generate an If-None-Match header field containing a list of those entity-tags when making a GET request; this allows recipient servers to send a 304 (Not Modified) response to indicate when one of those stored responses matches the selected representation.

If-None-Match can also be used with a value of "*" to prevent an unsafe request method (e.g., PUT) from inadvertently modifying an existing representation of the target resource when the client

believes that the resource does not have a current representation (Section 4.2.1 of [Part2]). This is a variation on the "lost update" problem that might arise if more than one client attempts to create an initial representation for the target resource.

An origin server that receives an If-None-Match header field MUST evaluate the condition prior to performing the method (Section 5). If the field-value is "*", the condition is false if the origin server has a current representation for the target resource. If the field-value is a list of entity-tags, the condition is false if one of the listed tags match the entity-tag of the selected representation.

An origin server MUST NOT perform the requested method if the condition evaluates to false; instead, the origin server MUST respond with either a) the 304 (Not Modified) status code if the request method is GET or HEAD; or, b) the 412 (Precondition Failed) status code for all other request methods.

Requirements on cache handling of a received If-None-Match header field are defined in Section 4.3.2 of [Part6].

3.3. If-Modified-Since

The "If-Modified-Since" header field makes a GET or HEAD request method conditional on the selected representation's modification date being more recent than the date provided in the field-value. Transfer of the selected representation's data is avoided if that data has not changed.

If-Modified-Since = HTTP-date

An example of the field is:

If-Modified-Since: Sat, 29 Oct 1994 19:43:31 GMT

A recipient MUST ignore If-Modified-Since if the request contains an If-None-Match header field; the condition in If-None-Match is considered to be a more accurate replacement for the condition in If-Modified-Since and the two are only combined for the sake of interoperating with older intermediaries that might not implement If-None-Match.

A recipient MUST ignore the If-Modified-Since header field if the received field-value is not a valid HTTP-date, or if the request method is neither GET nor HEAD.

A recipient MUST interpret an If-Modified-Since field-value's

timestamp in terms of the origin server's clock.

If-Modified-Since is typically used for two distinct purposes: 1) to allow efficient updates of a cached representation that does not have an entity-tag; and, 2) to limit the scope of a web traversal to resources that have recently changed.

When used for cache updates, a cache will typically use the value of the cached message's Last-Modified field to generate the field value of If-Modified-Since. This behavior is most interoperable for cases where clocks are poorly synchronized or when the server has chosen to only honor exact timestamp matches (due to a problem with Last-Modified dates that appear to go "back in time" when the origin server's clock is corrected or a representation is restored from an archived backup). However, caches occasionally generate the field value based on other data, such as the Date header field of the cached message or the local clock time that the message was received, particularly when the cached message does not contain a Last-Modified field.

When used for limiting the scope of retrieval to a recent time window, a user agent will generate an If-Modified-Since field value based on either its own local clock or a Date header field received from the server in a prior response. Origin servers that choose an exact timestamp match based on the selected representation's Last-Modified field will not be able to help the user agent limit its data transfers to only those changed during the specified window.

An origin server that receives an If-Modified-Since header field SHOULD evaluate the condition prior to performing the method (Section 5). The origin server SHOULD NOT perform the requested method if the selected representation's last modification date is earlier than or equal to the date provided in the field-value; instead, the origin server SHOULD generate a 304 (Not Modified) response, including only those metadata that are useful for identifying or updating a previously cached response.

Requirements on cache handling of a received If-Modified-Since header field are defined in Section 4.3.2 of [Part6].

3.4. If-Unmodified-Since

The "If-Unmodified-Since" header field makes the request method conditional on the selected representation's last modification date being earlier than or equal to the date provided in the field-value. This field accomplishes the same purpose as If-Match for cases where the user agent does not have an entity-tag for the representation.

If-Unmodified-Since = HTTP-date

An example of the field is:

If-Unmodified-Since: Sat, 29 Oct 1994 19:43:31 GMT

A recipient MUST ignore If-Unmodified-Since if the request contains an If-Match header field; the condition in If-Match is considered to be a more accurate replacement for the condition in If-Unmodified-Since and the two are only combined for the sake of interoperating with older intermediaries that might not implement If-Match.

A recipient MUST ignore the If-Unmodified-Since header field if the received field-value is not a valid HTTP-date.

A recipient MUST interpret an If-Unmodified-Since field-value's timestamp in terms of the origin server's clock.

If-Unmodified-Since is most often used with state-changing methods (e.g., POST, PUT, DELETE) to prevent accidental overwrites when multiple user agents might be acting in parallel on a resource that does not supply entity-tags with its representations (i.e., to prevent the "lost update" problem). It can also be used with safe methods to abort a request if the selected representation does not match one already stored (or partially stored) from a prior request.

An origin server that receives an If-Unmodified-Since header field MUST evaluate the condition prior to performing the method (Section 5). The origin server MUST NOT perform the requested method if the selected representation's last modification date is more recent than the date provided in the field-value; instead the origin server MUST respond with either: a) the 412 (Precondition Failed) status code; or, b) one of the 2xx (Successful) status codes if the origin server has verified that a state change is being requested and the final state is already reflected in the current state of the target resource (i.e., the change requested by the user agent has already succeeded, but the user agent might not be aware of that because the prior response message was lost or a compatible change was made by some other user agent). In the latter case, the origin server MUST NOT send a validator header field in the response unless it can verify that the request is a duplicate of an immediately prior change made by the same user agent.

The If-Unmodified-Since header field can be ignored by caches and intermediaries because it is not applicable to a stored response.

3.5. If-Range

The "If-Range" header field provides a special conditional request mechanism that is similar to the If-Match and If-Unmodified-Since header fields but instructs the recipient to ignore the Range header field if the validator doesn't match, resulting in transfer of the new selected representation instead of a 412 response. If-Range is defined in Section 3.2 of [Part5].

4. Status Code Definitions

4.1. 304 Not Modified

The 304 (Not Modified) status code indicates that a conditional GET or HEAD request has been received and would have resulted in a 200 (OK) response if it were not for the fact that the condition has evaluated to false. In other words, there is no need for the server to transfer a representation of the target resource because the request indicates that the client, which made the request conditional, already has a valid representation; the server is therefore redirecting the client to make use of that stored representation as if it were the payload of a 200 (OK) response.

The server generating a 304 response MUST generate any of the following header fields that would have been sent in a 200 (OK) response to the same request: Cache-Control, Content-Location, Date, ETag, Expires, and Vary.

Since the goal of a 304 response is to minimize information transfer when the recipient already has one or more cached representations, a sender SHOULD NOT generate representation metadata other than the above listed fields unless said metadata exists for the purpose of guiding cache updates (e.g., Last-Modified might be useful if the response does not have an ETag field).

Requirements on a cache that receives a 304 response are defined in Section 4.3.4 of [Part6]. If the conditional request originated with an outbound client, such as a user agent with its own cache sending a conditional GET to a shared proxy, then the proxy SHOULD forward the 304 response to that client.

A 304 response cannot contain a message-body; it is always terminated by the first empty line after the header fields.

4.2. 412 Precondition Failed

The 412 (Precondition Failed) status code indicates that one or more conditions given in the request header fields evaluated to false when

tested on the server. This response code allows the client to place preconditions on the current resource state (its current representations and metadata) and thus prevent the request method from being applied if the target resource is in an unexpected state.

5. Evaluation

Except when excluded below, a recipient cache or origin server **MUST** evaluate received request preconditions after it has successfully performed its normal request checks and just before it would perform the action associated with the request method. A server **MUST** ignore all received preconditions if its response to the same request without those conditions would have been a status code other than a 2xx or 412 (Precondition Failed). In other words, redirects and failures take precedence over the evaluation of preconditions in conditional requests.

A server that is not the origin server for the target resource and cannot act as a cache for requests on the target resource **MUST NOT** evaluate the conditional request header fields defined by this specification, and **MUST** forward them if the request is forwarded, since the generating client intends that they be evaluated by a server that can provide a current representation. Likewise, a server **MUST** ignore the conditional request header fields defined by this specification when received with a request method that does not involve the selection or modification of a selected representation, such as CONNECT, OPTIONS, or TRACE.

Conditional request header fields that are defined by extensions to HTTP might place conditions on all recipients, on the state of the target resource in general, or on a group of resources. For instance, the "If" header field in WebDAV can make a request conditional on various aspects of multiple resources, such as locks, if the recipient understands and implements that field ([RFC4918], Section 10.4).

Although conditional request header fields are defined as being usable with the HEAD method (to keep HEAD's semantics consistent with those of GET), there is no point in sending a conditional HEAD because a successful response is around the same size as a 304 (Not Modified) response and more useful than a 412 (Precondition Failed) response.

6. Precedence

When more than one conditional request header field is present in a request, the order in which the fields are evaluated becomes important. In practice, the fields defined in this document are

consistently implemented in a single, logical order, since "lost update" preconditions have more strict requirements than cache validation, a validated cache is more efficient than a partial response, and entity tags are presumed to be more accurate than date validators.

A recipient cache or origin server **MUST** evaluate the request preconditions defined by this specification in the following order:

1. When recipient is the origin server and If-Match is present, evaluate the If-Match precondition:
 - * if true, continue to step 3
 - * if false, respond 412 (Precondition Failed) unless it can be determined that the state-changing request has already succeeded (see Section 3.1)
2. When recipient is the origin server, If-Match is not present, and If-Unmodified-Since is present, evaluate the If-Unmodified-Since precondition:
 - * if true, continue to step 3
 - * if false, respond 412 (Precondition Failed) unless it can be determined that the state-changing request has already succeeded (see Section 3.4)
3. When If-None-Match is present, evaluate the If-None-Match precondition:
 - * if true, continue to step 5
 - * if false for GET/HEAD, respond 304 (Not Modified)
 - * if false for other methods, respond 412 (Precondition Failed)
4. When the method is GET or HEAD, If-None-Match is not present, and If-Modified-Since is present, evaluate the If-Modified-Since precondition:
 - * if true, continue to step 5
 - * if false, respond 304 (Not Modified)
5. When the method is GET and both Range and If-Range are present, evaluate the If-Range precondition:

- * if the validator matches and the Range specification is applicable to the selected representation, respond 206 (Partial Content) [Part5]

6. Otherwise,

- * all conditions are met, so perform the requested action and respond according to its success or failure.

Any extension to HTTP/1.1 that defines additional conditional request header fields ought to define its own expectations regarding the order for evaluating such fields in relation to those defined in this document and other conditionals that might be found in practice.

7. IANA Considerations

7.1. Status Code Registration

The HTTP Status Code Registry located at <http://www.iana.org/assignments/http-status-codes> shall be updated with the registrations below:

| Value | Description | Reference |
|-------|---------------------|-------------|
| 304 | Not Modified | Section 4.1 |
| 412 | Precondition Failed | Section 4.2 |

7.2. Header Field Registration

HTTP header fields are registered within the Message Header Field Registry maintained at <http://www.iana.org/assignments/message-headers/message-header-index.html>.

This document defines the following HTTP header fields, so their associated registry entries shall be updated according to the permanent registrations below (see [BCP90]):

| Header Field Name | Protocol | Status | Reference |
|---------------------|----------|----------|-------------|
| ETag | http | standard | Section 2.3 |
| If-Match | http | standard | Section 3.1 |
| If-Modified-Since | http | standard | Section 3.3 |
| If-None-Match | http | standard | Section 3.2 |
| If-Unmodified-Since | http | standard | Section 3.4 |
| Last-Modified | http | standard | Section 2.2 |

The change controller is: "IETF (iesg@ietf.org) - Internet Engineering Task Force".

8. Security Considerations

This section is meant to inform developers, information providers, and users of known security concerns specific to the HTTP conditional request mechanisms. More general security considerations are addressed in HTTP messaging [Part1] and semantics [Part2].

The validators defined by this specification are not intended to ensure the validity of a representation, guard against malicious changes, or detect man-in-the-middle attacks. At best, they enable more efficient cache updates and optimistic concurrent writes when all participants are behaving nicely. At worst, the conditions will fail and the client will receive a response that is no more harmful than an HTTP exchange without conditional requests.

An entity-tag can be abused in ways that create privacy risks. For example, a site might deliberately construct a semantically invalid entity-tag that is unique to the user or user agent, send it in a cacheable response with a long freshness time, and then read that entity-tag in later conditional requests as a means of re-identifying that user or user agent. Such an identifying tag would become a persistent identifier for as long as the user agent retained the original cache entry. User agents that cache representations ought to ensure that the cache is cleared or replaced whenever the user performs privacy-maintaining actions, such as clearing stored cookies or changing to a private browsing mode.

9. Acknowledgments

See Section 10 of [Part1].

10. References

10.1. Normative References

- [Part1] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", draft-ietf-httpbis-p1-messaging-26 (work in progress), February 2014.
- [Part2] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", draft-ietf-httpbis-p2-semantics-26 (work in progress), February 2014.
- [Part5] Fielding, R., Ed., Lafon, Y., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Range Requests", draft-ietf-httpbis-p5-range-26 (work in progress), February 2014.
- [Part6] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Caching", draft-ietf-httpbis-p6-cache-26 (work in progress), February 2014.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008.

10.2. Informative References

- [BCP90] Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, September 2004.
- [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.
- [RFC4918] Dusseault, L., Ed., "HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV)", RFC 4918, June 2007.

Appendix A. Changes from RFC 2616

The definition of validator weakness has been expanded and clarified. (Section 2.1)

Weak entity-tags are now allowed in all requests except range requests. (Sections 2.1 and 3.2)

The ETag header field ABNF has been changed to not use quoted-string, thus avoiding escaping issues. (Section 2.3)

ETag is defined to provide an entity tag for the selected representation, thereby clarifying what it applies to in various situations (such as a PUT response). (Section 2.3)

The precedence for evaluation of conditional requests has been defined. (Section 6)

Appendix B. Imported ABNF

The following core rules are included by reference, as defined in Appendix B.1 of [RFC5234]: ALPHA (letters), CR (carriage return), CRLF (CR LF), CTL (controls), DIGIT (decimal 0-9), DQUOTE (double quote), HEXDIG (hexadecimal 0-9/A-F/a-f), LF (line feed), OCTET (any 8-bit sequence of data), SP (space), and VCHAR (any visible US-ASCII character).

The rules below are defined in [Part1]:

| | |
|----------|---|
| OWS | = <OWS, defined in [Part1], Section 3.2.3> |
| obs-text | = <obs-text, defined in [Part1], Section 3.2.6> |

The rules below are defined in other parts:

| | |
|-----------|--|
| HTTP-date | = <HTTP-date, defined in [Part2], Section 7.1.1.1> |
|-----------|--|

Appendix C. Collected ABNF

In the collected ABNF below, list rules are expanded as per Section 1.2 of [Part1].

ETag = entity-tag

HTTP-date = <HTTP-date, defined in [Part2], Section 7.1.1.1>

If-Match = "*" / (*("," OWS) entity-tag *(OWS "," [OWS entity-tag]))

If-Modified-Since = HTTP-date

If-None-Match = "*" / (*("," OWS) entity-tag *(OWS "," [OWS entity-tag]))

If-Unmodified-Since = HTTP-date

Last-Modified = HTTP-date

OWS = <OWS, defined in [Part1], Section 3.2.3>

entity-tag = [weak] opaque-tag

etagc = "!" / %x23-7E ; '#'-'~'
/ obs-text

obs-text = <obs-text, defined in [Part1], Section 3.2.6>

opaque-tag = DQUOTE *etagc DQUOTE

weak = %x57.2F ; W/

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

Changes up to the IETF Last Call draft are summarized in <<http://tools.ietf.org/html/draft-ietf-httpbis-p4-conditional-24#appendix-D>>.

D.1. Since draft-ietf-httpbis-p4-conditional-24

Closed issues:

- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/518>>: "APPSDIR review of draft-ietf-httpbis-p4-conditional-24"

D.2. Since draft-ietf-httpbis-p4-conditional-25

Closed issues:

- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/538>>: "add 'stateless' to Abstract"
- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/542>>: "improve introduction of list rule"
- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/549>>: "augment security considerations with pointers to current research"

Index

- 3
 - 304 Not Modified (status code) 18
- 4
 - 412 Precondition Failed (status code) 18
- E
 - ETag header field 9
- G
 - Grammar
 - entity-tag 9
 - ETag 9
 - etagc 9
 - If-Match 13
 - If-Modified-Since 15
 - If-None-Match 14
 - If-Unmodified-Since 16
 - Last-Modified 7
 - opaque-tag 9
 - weak 9
- I
 - If-Match header field 13
 - If-Modified-Since header field 15
 - If-None-Match header field 14
 - If-Unmodified-Since header field 16
- L
 - Last-Modified header field 7
- M
 - metadata 5
- S
 - selected representation 4
- V
 - validator 5
 - strong 5
 - weak 5

Authors' Addresses

Roy T. Fielding (editor)
Adobe Systems Incorporated
345 Park Ave
San Jose, CA 95110
USA

EMail: fielding@gbiv.com
URI: <http://roy.gbiv.com/>

Julian F. Reschke (editor)
greenbytes GmbH
Hafenweg 16
Muenster, NW 48155
Germany

EMail: julian.reschke@greenbytes.de
URI: <http://greenbytes.de/tech/webdav/>

HTTPbis Working Group
Internet-Draft
Obsoletes: 2616 (if approved)
Intended status: Standards Track
Expires: August 10, 2014

R. Fielding, Ed.
Adobe
Y. Lafon, Ed.
W3C
J. Reschke, Ed.
greenbytes
February 6, 2014

Hypertext Transfer Protocol (HTTP/1.1): Range Requests
draft-ietf-httpbis-p5-range-26

Abstract

The Hypertext Transfer Protocol (HTTP) is a stateless application-level protocol for distributed, collaborative, hypertext information systems. This document defines range requests and the rules for constructing and combining responses to those requests.

Editorial Note (To be removed by RFC Editor)

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

The current issues list is at <http://tools.ietf.org/wg/httpbis/trac/report/3> and related documents (including fancy diffs) can be found at <http://tools.ietf.org/wg/httpbis/>.

The changes in this draft are summarized in Appendix E.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 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.

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.

Table of Contents

| | |
|--|----|
| 1. Introduction | 4 |
| 1.1. Conformance and Error Handling | 4 |
| 1.2. Syntax Notation | 4 |
| 2. Range Units | 4 |
| 2.1. Byte Ranges | 5 |
| 2.2. Other Range Units | 7 |
| 2.3. Accept-Ranges | 7 |
| 3. Range Requests | 7 |
| 3.1. Range | 7 |
| 3.2. If-Range | 9 |
| 4. Responses to a Range Request | 10 |
| 4.1. 206 Partial Content | 10 |
| 4.2. Content-Range | 12 |
| 4.3. Combining Ranges | 14 |
| 4.4. 416 Range Not Satisfiable | 15 |
| 5. IANA Considerations | 15 |
| 5.1. Range Unit Registry | 15 |
| 5.1.1. Procedure | 15 |
| 5.1.2. Registrations | 16 |
| 5.2. Status Code Registration | 16 |
| 5.3. Header Field Registration | 16 |
| 5.4. Internet Media Type Registration | 17 |
| 5.4.1. Internet Media Type multipart/byteranges | 17 |
| 6. Security Considerations | 18 |
| 6.1. Denial of Service Attacks using Range | 18 |
| 7. Acknowledgments | 18 |
| 8. References | 19 |
| 8.1. Normative References | 19 |
| 8.2. Informative References | 19 |
| Appendix A. Internet Media Type multipart/byteranges | 20 |
| Appendix B. Changes from RFC 2616 | 21 |
| Appendix C. Imported ABNF | 21 |
| Appendix D. Collected ABNF | 21 |
| Appendix E. Change Log (to be removed by RFC Editor before publication) | 23 |
| E.1. Since draft-ietf-httpbis-p5-range-24 | 23 |
| E.2. Since draft-ietf-httpbis-p5-range-25 | 23 |
| Index | 23 |

1. Introduction

Hypertext Transfer Protocol (HTTP) clients often encounter interrupted data transfers as a result of canceled requests or dropped connections. When a client has stored a partial representation, it is desirable to request the remainder of that representation in a subsequent request rather than transfer the entire representation. Likewise, devices with limited local storage might benefit from being able to request only a subset of a larger representation, such as a single page of a very large document, or the dimensions of an embedded image.

This document defines HTTP/1.1 range requests, partial responses, and the multipart/byteranges media type. Range requests are an OPTIONAL feature of HTTP, designed so that recipients not implementing this feature (or not supporting it for the target resource) can respond as if it is a normal GET request without impacting interoperability. Partial responses are indicated by a distinct status code to not be mistaken for full responses by caches that might not implement the feature.

Although the range request mechanism is designed to allow for extensible range types, this specification only defines requests for byte ranges.

1.1. Conformance and Error Handling

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].

Conformance criteria and considerations regarding error handling are defined in Section 2.5 of [Part1].

1.2. Syntax Notation

This specification uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234] with a list extension, defined in Section 7 of [Part1], that allows for compact definition of comma-separated lists using a '#' operator (similar to how the '*' operator indicates repetition). Appendix C describes rules imported from other documents. Appendix D shows the collected grammar with all list operators expanded to standard ABNF notation.

2. Range Units

A representation can be partitioned into subranges according to various structural units, depending on the structure inherent in the

representation's media type. This "range unit" is used in the Accept-Ranges (Section 2.3) response header field to advertise support for range requests, the Range (Section 3.1) request header field to delineate the parts of a representation that are requested, and the Content-Range (Section 4.2) payload header field to describe which part of a representation is being transferred.

range-unit = bytes-unit / other-range-unit

2.1. Byte Ranges

Since representation data is transferred in payloads as a sequence of octets, a byte range is a meaningful substructure for any representation transferable over HTTP (Section 3 of [Part2]). The "bytes" range unit is defined for expressing subranges of the data's octet sequence.

bytes-unit = "bytes"

A byte range request can specify a single range of bytes, or a set of ranges within a single representation.

```
byte-ranges-specifier = bytes-unit "=" byte-range-set
byte-range-set       = 1#( byte-range-spec / suffix-byte-range-spec )
byte-range-spec      = first-byte-pos "-" [ last-byte-pos ]
first-byte-pos       = 1*DIGIT
last-byte-pos        = 1*DIGIT
```

The first-byte-pos value in a byte-range-spec gives the byte-offset of the first byte in a range. The last-byte-pos value gives the byte-offset of the last byte in the range; that is, the byte positions specified are inclusive. Byte offsets start at zero.

Examples of byte-ranges-specifier values:

- o The first 500 bytes (byte offsets 0-499, inclusive):

bytes=0-499

- o The second 500 bytes (byte offsets 500-999, inclusive):

bytes=500-999

A byte-range-spec is invalid if the last-byte-pos value is present and less than the first-byte-pos.

A client can limit the number of bytes requested without knowing the size of the selected representation. If the last-byte-pos value is

absent, or if the value is greater than or equal to the current length of the representation data, the byte range is interpreted as the remainder of the representation (i.e., the server replaces the value of last-byte-pos with a value that is one less than the current length of the selected representation).

A client can request the last N bytes of the selected representation using a suffix-byte-range-spec.

```
suffix-byte-range-spec = "-" suffix-length
suffix-length = 1*DIGIT
```

If the selected representation is shorter than the specified suffix-length, the entire representation is used.

Additional examples, assuming a representation of length 10000:

- o The final 500 bytes (byte offsets 9500-9999, inclusive):

```
bytes=-500
```

Or:

```
bytes=9500-
```

- o The first and last bytes only (bytes 0 and 9999):

```
bytes=0-0,-1
```

- o Other valid (but not canonical) specifications of the second 500 bytes (byte offsets 500-999, inclusive):

```
bytes=500-600,601-999
```

```
bytes=500-700,601-999
```

If a valid byte-range-set includes at least one byte-range-spec with a first-byte-pos that is less than the current length of the representation, or at least one suffix-byte-range-spec with a non-zero suffix-length, then the byte-range-set is satisfiable. Otherwise, the byte-range-set is unsatisfiable.

In the byte range syntax, first-byte-pos, last-byte-pos, and suffix-length are expressed as decimal number of octets. Since there is no predefined limit to the length of a payload, recipients MUST anticipate potentially large decimal numerals and prevent parsing errors due to integer conversion overflows.

2.2. Other Range Units

Range units are intended to be extensible. New range units ought to be registered with IANA, as defined in Section 5.1.

other-range-unit = token

2.3. Accept-Ranges

The "Accept-Ranges" header field allows a server to indicate that it supports range requests for the target resource.

Accept-Ranges = acceptable-ranges
acceptable-ranges = 1#range-unit / "none"

An origin server that supports byte-range requests for a given target resource MAY send

Accept-Ranges: bytes

to indicate what range units are supported. A client MAY generate range requests without having received this header field for the resource involved. Range units are defined in Section 2.

A server that does not support any kind of range request for the target resource MAY send

Accept-Ranges: none

to advise the client not to attempt a range request.

3. Range Requests

3.1. Range

The "Range" header field on a GET request modifies the method semantics to request transfer of only one or more subranges of the selected representation data, rather than the entire selected representation data.

Range = byte-ranges-specifier / other-ranges-specifier
other-ranges-specifier = other-range-unit "=" other-range-set
other-range-set = 1*VCHAR

A server MAY ignore the Range header field. However, origin servers and intermediate caches ought to support byte ranges when possible, since Range supports efficient recovery from partially failed transfers and partial retrieval of large representations. A server

MUST ignore a Range header field received with a request method other than GET.

An origin server MUST ignore a Range header field that contains a range unit it does not understand. A proxy MAY discard a Range header field that contains a range unit it does not understand.

A server that supports range requests MAY ignore or reject a Range header field that consists of more than two overlapping ranges, or a set of many small ranges that are not listed in ascending order, since both are indications of either a broken client or a deliberate denial of service attack (Section 6.1). A client SHOULD NOT request multiple ranges that are inherently less efficient to process and transfer than a single range that encompasses the same data.

A client that is requesting multiple ranges SHOULD list those ranges in ascending order (the order in which they would typically be received in a complete representation) unless there is a specific need to request a later part earlier. For example, a user agent processing a large representation with an internal catalog of parts might need to request later parts first, particularly if the representation consists of pages stored in reverse order and the user agent wishes to transfer one page at a time.

The Range header field is evaluated after evaluating the precondition header fields defined in [Part4], and only if the result in absence of the Range header field would be a 200 (OK) response. In other words, Range is ignored when a conditional GET would result in a 304 (Not Modified) response.

The If-Range header field (Section 3.2) can be used as a precondition to applying the Range header field.

If all of the preconditions are true, the server supports the Range header field for the target resource, and the specified range(s) are valid and satisfiable (as defined in Section 2.1), the server SHOULD send a 206 (Partial Content) response with a payload containing one or more partial representations that correspond to the satisfiable ranges requested, as defined in Section 4.

If all of the preconditions are true, the server supports the Range header field for the target resource, and the specified range(s) are invalid or unsatisfiable, the server SHOULD send a 416 (Range Not Satisfiable) response.

3.2. If-Range

If a client has a partial copy of a representation and wishes to have an up-to-date copy of the entire representation, it could use the Range header field with a conditional GET (using either or both of If-Unmodified-Since and If-Match.) However, if the precondition fails because the representation has been modified, the client would then have to make a second request to obtain the entire current representation.

The "If-Range" header field allows a client to "short-circuit" the second request. Informally, its meaning is: if the representation is unchanged, send me the part(s) that I am requesting in Range; otherwise, send me the entire representation.

If-Range = entity-tag / HTTP-date

A client MUST NOT generate an If-Range header field in a request that does not contain a Range header field. A server MUST ignore an If-Range header field received in a request that does not contain a Range header field. An origin server MUST ignore an If-Range header field received in a request for a target resource that does not support Range requests.

A client MUST NOT generate an If-Range header field containing an entity-tag that is marked as weak. A client MUST NOT generate an If-Range header field containing an HTTP-date unless the client has no entity-tag for the corresponding representation and the date is a strong validator in the sense defined by Section 2.2.2 of [Part4].

A server that evaluates an If-Range precondition MUST use the strong comparison function when comparing entity-tags (Section 2.3.2 of [Part4]) and MUST evaluate the condition as false if an HTTP-date validator is provided that is not a strong validator in the sense defined by Section 2.2.2 of [Part4]. A valid entity-tag can be distinguished from a valid HTTP-date by examining the first two characters for a DQUOTE.

If the validator given in the If-Range header field matches the current validator for the selected representation of the target resource, then the server SHOULD process the Range header field as requested. If the validator does not match, the server MUST ignore the Range header field. Note that this comparison by exact match, including when the validator is an HTTP-date, differs from the "earlier than or equal to" comparison used when evaluating an If-Unmodified-Since conditional.

4. Responses to a Range Request

4.1. 206 Partial Content

The 206 (Partial Content) status code indicates that the server is successfully fulfilling a range request for the target resource by transferring one or more parts of the selected representation that correspond to the satisfiable ranges found in the request's Range header field (Section 3.1).

If a single part is being transferred, the server generating the 206 response MUST generate a Content-Range header field, describing what range of the selected representation is enclosed, and a payload consisting of the range. For example:

```
HTTP/1.1 206 Partial Content
Date: Wed, 15 Nov 1995 06:25:24 GMT
Last-Modified: Wed, 15 Nov 1995 04:58:08 GMT
Content-Range: bytes 21010-47021/47022
Content-Length: 26012
Content-Type: image/gif
```

... 26012 bytes of partial image data ...

If multiple parts are being transferred, the server generating the 206 response MUST generate a "multipart/byteranges" payload, as defined in Appendix A, and a Content-Type header field containing the multipart/byteranges media type and its required boundary parameter. To avoid confusion with single part responses, a server MUST NOT generate a Content-Range header field in the HTTP header section of a multiple part response (this field will be sent in each part instead).

Within the header area of each body part in the multipart payload, the server MUST generate a Content-Range header field corresponding to the range being enclosed in that body part. If the selected representation would have had a Content-Type header field in a 200 (OK) response, the server SHOULD generate that same Content-Type field in the header area of each body part. For example:

```
HTTP/1.1 206 Partial Content
Date: Wed, 15 Nov 1995 06:25:24 GMT
Last-Modified: Wed, 15 Nov 1995 04:58:08 GMT
Content-Length: 1741
Content-Type: multipart/byteranges; boundary=THIS_STRING_SEPARATES

--THIS_STRING_SEPARATES
Content-Type: application/pdf
Content-Range: bytes 500-999/8000

...the first range...
--THIS_STRING_SEPARATES
Content-Type: application/pdf
Content-Range: bytes 7000-7999/8000

...the second range
--THIS_STRING_SEPARATES--
```

When multiple ranges are requested, a server MAY coalesce any of the ranges that overlap, or that are separated by a gap that is smaller than the overhead of sending multiple parts, regardless of the order in which the corresponding byte-range-spec appeared in the received Range header field. Since the typical overhead between parts of a multipart/byteranges payload is around 80 bytes, depending on the selected representation's media type and the chosen boundary parameter length, it can be less efficient to transfer many small disjoint parts than it is to transfer the entire selected representation.

A server MUST NOT generate a multipart response to a request for a single range, since a client that does not request multiple parts might not support multipart responses. However, a server MAY generate a multipart/byteranges payload with only a single body part if multiple ranges were requested and only one range was found to be satisfiable or only one range remained after coalescing. A client that cannot process a multipart/byteranges response MUST NOT generate a request that asks for multiple ranges.

When a multipart response payload is generated, the server SHOULD send the parts in the same order that the corresponding byte-range-spec appeared in the received Range header field, excluding those ranges that were deemed unsatisfiable or that were coalesced into other ranges. A client that receives a multipart response MUST inspect the Content-Range header field present in each body part in order to determine which range is contained in that body part; a client cannot rely on receiving the same ranges that it requested, nor the same order that it requested.

When a 206 response is generated, the server MUST generate the following header fields, in addition to those required above, if the field would have been sent in a 200 (OK) response to the same request: Date, Cache-Control, ETag, Expires, Content-Location, and Vary.

If a 206 is generated in response to a request with an If-Range header field, the sender SHOULD NOT generate other representation header fields beyond those required above, because the client is understood to already have a prior response containing those header fields. Otherwise, the sender MUST generate all of the representation header fields that would have been sent in a 200 (OK) response to the same request.

A 206 response is cacheable by default; i.e., unless otherwise indicated by explicit cache controls (see Section 4.2.2 of [Part6]).

4.2. Content-Range

The "Content-Range" header field is sent in a single part 206 (Partial Content) response to indicate the partial range of the selected representation enclosed as the message payload, sent in each part of a multipart 206 response to indicate the range enclosed within each body part, and sent in 416 (Range Not Satisfiable) responses to provide information about the selected representation.

```
Content-Range      = byte-content-range
                   / other-content-range

byte-content-range = bytes-unit SP
                   ( byte-range-resp / unsatisfied-range )

byte-range-resp    = byte-range "/" ( complete-length / "*" )
byte-range         = first-byte-pos "-" last-byte-pos
unsatisfied-range  = "*" / complete-length

complete-length    = 1 *DIGIT

other-content-range = other-range-unit SP other-range-resp
other-range-resp    = *CHAR
```

If a 206 (Partial Content) response contains a Content-Range header field with a range unit (Section 2) that the recipient does not understand, the recipient MUST NOT attempt to recombine it with a stored representation. A proxy that receives such a message SHOULD forward it downstream.

For byte ranges, a sender SHOULD indicate the complete length of the

representation from which the range has been extracted, unless the complete length is unknown or difficult to determine. An asterisk character ("*") in place of the complete-length indicates that the representation length was unknown when the header field was generated.

The following example illustrates when the complete length of the selected representation is known by the sender to be 1234 bytes:

```
Content-Range: bytes 42-1233/1234
```

and this second example illustrates when the complete length is unknown:

```
Content-Range: bytes 42-1233/*
```

A Content-Range field value is invalid if it contains a byte-range-resp that has a last-byte-pos value less than its first-byte-pos value, or a complete-length value less than or equal to its last-byte-pos value. The recipient of an invalid Content-Range MUST NOT attempt to recombine the received content with a stored representation.

A server generating a 416 (Range Not Satisfiable) response to a byte range request SHOULD send a Content-Range header field with an unsatisfied-range value, as in the following example:

```
Content-Range: bytes */1234
```

The complete-length in a 416 response indicates the current length of the selected representation.

The "Content-Range" header field has no meaning for status codes that do not explicitly describe its semantic. For this specification, only the 206 (Partial Content) and 416 (Range Not Satisfiable) status codes describe a meaning for Content-Range.

The following are examples of Content-Range values in which the selected representation contains a total of 1234 bytes:

- o The first 500 bytes:

```
Content-Range: bytes 0-499/1234
```

- o The second 500 bytes:

```
Content-Range: bytes 500-999/1234
```

- o All except for the first 500 bytes:

Content-Range: bytes 500-1233/1234

- o The last 500 bytes:

Content-Range: bytes 734-1233/1234

4.3. Combining Ranges

A response might transfer only a subrange of a representation if the connection closed prematurely or if the request used one or more Range specifications. After several such transfers, a client might have received several ranges of the same representation. These ranges can only be safely combined if they all have in common the same strong validator (Section 2.1 of [Part4]).

A client that has received multiple partial responses to GET requests on a target resource MAY combine those responses into a larger continuous range if they share the same strong validator.

If the most recent response is an incomplete 200 (OK) response, then the header fields of that response are used for any combined response and replace those of the matching stored responses.

If the most recent response is a 206 (Partial Content) response and at least one of the matching stored responses is a 200 (OK), then the combined response header fields consist of the most recent 200 response's header fields. If all of the matching stored responses are 206 responses, then the stored response with the most recent header fields is used as the source of header fields for the combined response, except that the client MUST use other header fields provided in the new response, aside from Content-Range, to replace all instances of the corresponding header fields in the stored response.

The combined response message body consists of the union of partial content ranges in the new response and each of the selected responses. If the union consists of the entire range of the representation, then the client MUST process the combined response as if it were a complete 200 (OK) response, including a Content-Length header field that reflects the complete length. Otherwise, the client MUST process the set of continuous ranges as one of the following: an incomplete 200 (OK) response if the combined response is a prefix of the representation, a single 206 (Partial Content) response containing a multipart/byteranges body, or multiple 206 (Partial Content) responses, each with one continuous range that is indicated by a Content-Range header field.

4.4. 416 Range Not Satisfiable

The 416 (Range Not Satisfiable) status code indicates that none of the ranges in the request's Range header field (Section 3.1) overlap the current extent of the selected resource or that the set of ranges requested has been rejected due to invalid ranges or an excessive request of small or overlapping ranges.

For byte ranges, failing to overlap the current extent means that the first-byte-pos of all of the byte-range-spec values were greater than the current length of the selected representation. When this status code is generated in response to a byte range request, the sender SHOULD generate a Content-Range header field specifying the current length of the selected representation (Section 4.2).

For example:

```
HTTP/1.1 416 Range Not Satisfiable
Date: Fri, 20 Jan 2012 15:41:54 GMT
Content-Range: bytes */47022
```

Note: Because servers are free to ignore Range, many implementations will simply respond with the entire selected representation in a 200 (OK) response. That is partly because most clients are prepared to receive a 200 (OK) to complete the task (albeit less efficiently) and partly because clients might not stop making an invalid partial request until they have received a complete representation. Thus, clients cannot depend on receiving a 416 (Range Not Satisfiable) response even when it is most appropriate.

5. IANA Considerations

5.1. Range Unit Registry

The HTTP Range Unit Registry defines the name space for the range unit names and refers to their corresponding specifications. The registry will be created and maintained at (the suggested URI) <http://www.iana.org/assignments/http-parameters>.

5.1.1. Procedure

Registration of an HTTP Range Unit MUST include the following fields:

- o Name
- o Description

- o Pointer to specification text

Values to be added to this name space require IETF Review (see [RFC5226], Section 4.1).

5.1.2. Registrations

The initial HTTP Range Unit Registry shall contain the registrations below:

| Range Unit Name | Description | Reference |
|-----------------|---|-------------|
| bytes | a range of octets | Section 2.1 |
| none | reserved as keyword, indicating no ranges are supported | Section 2.3 |

The change controller is: "IETF (iesg@ietf.org) - Internet Engineering Task Force".

5.2. Status Code Registration

The HTTP Status Code Registry located at <http://www.iana.org/assignments/http-status-codes> shall be updated with the registrations below:

| Value | Description | Reference |
|-------|-----------------------|-------------|
| 206 | Partial Content | Section 4.1 |
| 416 | Range Not Satisfiable | Section 4.4 |

5.3. Header Field Registration

HTTP header fields are registered within the Message Header Field Registry maintained at <http://www.iana.org/assignments/message-headers/message-header-index.html>.

This document defines the following HTTP header fields, so their associated registry entries shall be updated according to the permanent registrations below (see [BCP90]):

| Header Field Name | Protocol | Status | Reference |
|-------------------|----------|----------|-------------|
| Accept-Ranges | http | standard | Section 2.3 |
| Content-Range | http | standard | Section 4.2 |
| If-Range | http | standard | Section 3.2 |
| Range | http | standard | Section 3.1 |

The change controller is: "IETF (iesg@ietf.org) - Internet Engineering Task Force".

5.4. Internet Media Type Registration

IANA maintains the registry of Internet media types [BCP13] at <http://www.iana.org/assignments/media-types>.

This document serves as the specification for the Internet media type "multipart/byteranges". The following is to be registered with IANA.

5.4.1. Internet Media Type multipart/byteranges

Type name: multipart

Subtype name: byteranges

Required parameters: boundary

Optional parameters: N/A

Encoding considerations: only "7bit", "8bit", or "binary" are permitted

Security considerations: see Section 6

Interoperability considerations: N/A

Published specification: This specification (see Appendix A).

Applications that use this media type: HTTP components supporting multiple ranges in a single request.

Fragment identifier considerations: N/A

Additional information:

Deprecated alias names for this type: N/A

Magic number(s): N/A

File extension(s): N/A

Macintosh file type code(s): N/A

Person and email address to contact for further information: See
Authors Section.

Intended usage: COMMON

Restrictions on usage: N/A

Author: See Authors Section.

Change controller: IESG

6. Security Considerations

This section is meant to inform developers, information providers, and users of known security concerns specific to the HTTP range request mechanisms. More general security considerations are addressed in HTTP messaging [Part1] and semantics [Part2].

6.1. Denial of Service Attacks using Range

Unconstrained multiple range requests are susceptible to denial of service attacks because the effort required to request many overlapping ranges of the same data is tiny compared to the time, memory, and bandwidth consumed by attempting to serve the requested data in many parts. Servers ought to ignore, coalesce, or reject egregious range requests, such as requests for more than two overlapping ranges or for many small ranges in a single set, particularly when the ranges are requested out of order for no apparent reason. Multipart range requests are not designed to support random access.

7. Acknowledgments

See Section 10 of [Part1].

8. References

8.1. Normative References

- [Part1] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", draft-ietf-httpbis-p1-messaging-26 (work in progress), February 2014.
- [Part2] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", draft-ietf-httpbis-p2-semantics-26 (work in progress), February 2014.
- [Part4] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests", draft-ietf-httpbis-p4-conditional-26 (work in progress), February 2014.
- [Part6] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Caching", draft-ietf-httpbis-p6-cache-26 (work in progress), February 2014.
- [RFC2046] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types", RFC 2046, November 1996.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008.

8.2. Informative References

- [BCP13] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, January 2013.
- [BCP90] Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, September 2004.
- [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.
- [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, February 2008.

May 2008.

Appendix A. Internet Media Type multipart/byteranges

When a 206 (Partial Content) response message includes the content of multiple ranges, they are transmitted as body parts in a multipart message body ([RFC2046], Section 5.1) with the media type of "multipart/byteranges".

The multipart/byteranges media type includes one or more body parts, each with its own Content-Type and Content-Range fields. The required boundary parameter specifies the boundary string used to separate each body part.

Implementation Notes:

1. Additional CRLFs might precede the first boundary string in the body.
2. Although [RFC2046] permits the boundary string to be quoted, some existing implementations handle a quoted boundary string incorrectly.
3. A number of clients and servers were coded to an early draft of the byteranges specification that used a media type of multipart/x-byteranges, which is almost (but not quite) compatible with this type.

Despite the name, the "multipart/byteranges" media type is not limited to byte ranges. The following example uses an "exampleunit" range unit:

```
HTTP/1.1 206 Partial Content
Date: Tue, 14 Nov 1995 06:25:24 GMT
Last-Modified: Tue, 14 July 04:58:08 GMT
Content-Length: 2331785
Content-Type: multipart/byteranges; boundary=THIS_STRING_SEPARATES

--THIS_STRING_SEPARATES
Content-Type: video/example
Content-Range: exampleunit 1.2-4.3/25

...the first range...
--THIS_STRING_SEPARATES
Content-Type: video/example
Content-Range: exampleunit 11.2-14.3/25

...the second range
```

--THIS_STRING_SEPARATES--

Appendix B. Changes from RFC 2616

Servers are given more leeway in how they respond to a range request, in order to mitigate abuse by malicious (or just greedy) clients. (Section 3.1)

A weak validator cannot be used in a 206 response. (Section 4.1)

The Content-Range header field only has meaning when the status code explicitly defines its use. (Section 4.2)

This specification introduces a Range Unit Registry. (Section 5.1)

multipart/byteranges can consist of a single part. (Appendix A)

Appendix C. Imported ABNF

The following core rules are included by reference, as defined in Appendix B.1 of [RFC5234]: ALPHA (letters), CR (carriage return), CRLF (CR LF), CTL (controls), DIGIT (decimal 0-9), DQUOTE (double quote), HEXDIG (hexadecimal 0-9/A-F/a-f), LF (line feed), OCTET (any 8-bit sequence of data), SP (space), and VCHAR (any visible US-ASCII character).

Note that all rules derived from token are to be compared case-insensitively, like range-unit and acceptable-ranges.

The rules below are defined in [Part1]:

OWS = <OWS, defined in [Part1], Section 3.2.3>
token = <token, defined in [Part1], Section 3.2.6>

The rules below are defined in other parts:

HTTP-date = <HTTP-date, defined in [Part2], Section 7.1.1.1>
entity-tag = <entity-tag, defined in [Part4], Section 2.3>

Appendix D. Collected ABNF

In the collected ABNF below, list rules are expanded as per Section 1.2 of [Part1].

Accept-Ranges = acceptable-ranges

Content-Range = byte-content-range / other-content-range

HTTP-date = <HTTP-date, defined in [Part2], Section 7.1.1.1>

If-Range = entity-tag / HTTP-date

OWS = <OWS, defined in [Part1], Section 3.2.3>

Range = byte-ranges-specifier / other-ranges-specifier

acceptable-ranges = (*("," OWS) range-unit *(OWS "," [OWS range-unit])) / "none"

byte-content-range = bytes-unit SP (byte-range-resp / unsatisfied-range)

byte-range = first-byte-pos "-" last-byte-pos

byte-range-resp = byte-range "/" (complete-length / "*")

byte-range-set = *("," OWS) (byte-range-spec / suffix-byte-range-spec) *(OWS "," [OWS (byte-range-spec / suffix-byte-range-spec)])

byte-range-spec = first-byte-pos "-" [last-byte-pos]

byte-ranges-specifier = bytes-unit "=" byte-range-set

bytes-unit = "bytes"

complete-length = 1*DIGIT

entity-tag = <entity-tag, defined in [Part4], Section 2.3>

first-byte-pos = 1*DIGIT

last-byte-pos = 1*DIGIT

other-content-range = other-range-unit SP other-range-resp

other-range-resp = *CHAR

other-range-set = 1*VCHAR

other-range-unit = token

other-ranges-specifier = other-range-unit "=" other-range-set

range-unit = bytes-unit / other-range-unit

suffix-byte-range-spec = "-" suffix-length

suffix-length = 1*DIGIT

token = <token, defined in [Part1], Section 3.2.6>

unsatisfied-range = "*" / complete-length

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

Changes up to the IETF Last Call draft are summarized in <http://tools.ietf.org/html/draft-ietf-httpbis-p5-range-24#appendix-E>.

E.1. Since draft-ietf-httpbis-p5-range-24

Closed issues:

- o <http://tools.ietf.org/wg/httpbis/trac/ticket/506>: "APPSDIR review of draft-ietf-httpbis-p5-range-24"
- o <http://tools.ietf.org/wg/httpbis/trac/ticket/507>: "integer value parsing"
- o <http://tools.ietf.org/wg/httpbis/trac/ticket/508>: "broken sentence in description of 206"

E.2. Since draft-ietf-httpbis-p5-range-25

Closed issues:

- o <http://tools.ietf.org/wg/httpbis/trac/ticket/526>: "check media type registration templates"
- o <http://tools.ietf.org/wg/httpbis/trac/ticket/527>: "use of CHAR for other-range-set"
- o <http://tools.ietf.org/wg/httpbis/trac/ticket/538>: "add 'stateless' to Abstract"
- o <http://tools.ietf.org/wg/httpbis/trac/ticket/542>: "improve introduction of list rule"
- o <http://tools.ietf.org/wg/httpbis/trac/ticket/549>: "augment security considerations with pointers to current research"

Index

| | | |
|---|---|----|
| 2 | 206 Partial Content (status code) | 10 |
| 4 | 416 Range Not Satisfiable (status code) | 15 |
| A | Accept-Ranges header field | 7 |

- C
 - Content-Range header field 12
- G
 - Grammar
 - Accept-Ranges 7
 - acceptable-ranges 7
 - byte-content-range 12
 - byte-range 12
 - byte-range-resp 12
 - byte-range-set 5
 - byte-range-spec 5
 - byte-ranges-specifier 5
 - bytes-unit 5
 - complete-length 12
 - Content-Range 12
 - first-byte-pos 5
 - If-Range 9
 - last-byte-pos 5
 - other-content-range 12
 - other-range-resp 12
 - other-range-unit 5, 7
 - Range 7
 - range-unit 5
 - ranges-specifier 5
 - suffix-byte-range-spec 6
 - suffix-length 6
 - unsatisfied-range 12
- I
 - If-Range header field 9
- M
 - Media Type
 - multipart/byteranges 17, 20
 - multipart/x-byteranges 20
 - multipart/byteranges Media Type 17, 20
 - multipart/x-byteranges Media Type 20
- R
 - Range header field 7

Authors' Addresses

Roy T. Fielding (editor)
Adobe Systems Incorporated
345 Park Ave
San Jose, CA 95110
USA

EMail: fielding@gbiv.com
URI: <http://roy.gbiv.com/>

Yves Lafon (editor)
World Wide Web Consortium
W3C / ERCIM
2004, rte des Lucioles
Sophia-Antipolis, AM 06902
France

EMail: ylafon@w3.org
URI: <http://www.raubacapeu.net/people/yves/>

Julian F. Reschke (editor)
greenbytes GmbH
Hafenweg 16
Muenster, NW 48155
Germany

EMail: julian.reschke@greenbytes.de
URI: <http://greenbytes.de/tech/webdav/>

HTTPbis Working Group
Internet-Draft
Obsoletes: 2616 (if approved)
Intended status: Standards Track
Expires: August 10, 2014

R. Fielding, Ed.
Adobe
M. Nottingham, Ed.
Akamai
J. Reschke, Ed.
greenbytes
February 6, 2014

Hypertext Transfer Protocol (HTTP/1.1): Caching
draft-ietf-httpbis-p6-cache-26

Abstract

The Hypertext Transfer Protocol (HTTP) is a stateless application-level protocol for distributed, collaborative, hypertext information systems. This document defines HTTP caches and the associated header fields that control cache behavior or indicate cacheable response messages.

Editorial Note (To be removed by RFC Editor)

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

The current issues list is at <http://tools.ietf.org/wg/httpbis/trac/report/3> and related documents (including fancy diffs) can be found at <http://tools.ietf.org/wg/httpbis/>.

The changes in this draft are summarized in Appendix 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 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.

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.

Table of Contents

| | |
|--|----|
| 1. Introduction | 4 |
| 1.1. Conformance and Error Handling | 4 |
| 1.2. Syntax Notation | 4 |
| 1.2.1. Delta Seconds | 5 |
| 2. Overview of Cache Operation | 5 |
| 3. Storing Responses in Caches | 6 |
| 3.1. Storing Incomplete Responses | 7 |
| 3.2. Storing Responses to Authenticated Requests | 7 |
| 3.3. Combining Partial Content | 8 |
| 4. Constructing Responses from Caches | 8 |
| 4.1. Calculating Secondary Keys with Vary | 9 |
| 4.2. Freshness | 10 |
| 4.2.1. Calculating Freshness Lifetime | 12 |
| 4.2.2. Calculating Heuristic Freshness | 12 |
| 4.2.3. Calculating Age | 13 |
| 4.2.4. Serving Stale Responses | 15 |
| 4.3. Validation | 15 |

| | | |
|-------------|--|----|
| 4.3.1. | Sending a Validation Request | 15 |
| 4.3.2. | Handling a Received Validation Request | 16 |
| 4.3.3. | Handling a Validation Response | 17 |
| 4.3.4. | Freshening Stored Responses upon Validation | 18 |
| 4.3.5. | Freshening Responses via HEAD | 19 |
| 4.4. | Invalidation | 19 |
| 5. | Header Field Definitions | 20 |
| 5.1. | Age | 20 |
| 5.2. | Cache-Control | 21 |
| 5.2.1. | Request Cache-Control Directives | 21 |
| 5.2.2. | Response Cache-Control Directives | 23 |
| 5.2.3. | Cache Control Extensions | 26 |
| 5.3. | Expires | 27 |
| 5.4. | Pragma | 28 |
| 5.5. | Warning | 29 |
| 5.5.1. | Warning: 110 - "Response is Stale" | 30 |
| 5.5.2. | Warning: 111 - "Revalidation Failed" | 31 |
| 5.5.3. | Warning: 112 - "Disconnected Operation" | 31 |
| 5.5.4. | Warning: 113 - "Heuristic Expiration" | 31 |
| 5.5.5. | Warning: 199 - "Miscellaneous Warning" | 31 |
| 5.5.6. | Warning: 214 - "Transformation Applied" | 31 |
| 5.5.7. | Warning: 299 - "Miscellaneous Persistent Warning" | 31 |
| 6. | History Lists | 31 |
| 7. | IANA Considerations | 32 |
| 7.1. | Cache Directive Registry | 32 |
| 7.1.1. | Procedure | 32 |
| 7.1.2. | Considerations for New Cache Control Directives | 32 |
| 7.1.3. | Registrations | 32 |
| 7.2. | Warn Code Registry | 33 |
| 7.2.1. | Procedure | 33 |
| 7.2.2. | Registrations | 33 |
| 7.3. | Header Field Registration | 34 |
| 8. | Security Considerations | 34 |
| 9. | Acknowledgments | 35 |
| 10. | References | 35 |
| 10.1. | Normative References | 35 |
| 10.2. | Informative References | 36 |
| Appendix A. | Changes from RFC 2616 | 36 |
| Appendix B. | Imported ABNF | 38 |
| Appendix C. | Collected ABNF | 38 |
| Appendix D. | Change Log (to be removed by RFC Editor before publication) | 39 |
| D.1. | Since draft-ietf-httpbis-p6-cache-24 | 40 |
| D.2. | Since draft-ietf-httpbis-p6-cache-25 | 40 |
| Index | | 40 |

1. Introduction

HTTP is typically used for distributed information systems, where performance can be improved by the use of response caches. This document defines aspects of HTTP/1.1 related to caching and reusing response messages.

An HTTP cache is a local store of response messages and the subsystem that controls storage, retrieval, and deletion of messages in it. A cache stores cacheable responses in order to reduce the response time and network bandwidth consumption on future, equivalent requests. Any client or server MAY employ a cache, though a cache cannot be used by a server that is acting as a tunnel.

A shared cache is a cache that stores responses to be reused by more than one user; shared caches are usually (but not always) deployed as a part of an intermediary. A private cache, in contrast, is dedicated to a single user; often, they are deployed as a component of a user agent.

The goal of caching in HTTP/1.1 is to significantly improve performance by reusing a prior response message to satisfy a current request. A stored response is considered "fresh", as defined in Section 4.2, if the response can be reused without "validation" (checking with the origin server to see if the cached response remains valid for this request). A fresh response can therefore reduce both latency and network overhead each time it is reused. When a cached response is not fresh, it might still be reusable if it can be freshened by validation (Section 4.3) or if the origin is unavailable (Section 4.2.4).

1.1. Conformance and Error Handling

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].

Conformance criteria and considerations regarding error handling are defined in Section 2.5 of [Part1].

1.2. Syntax Notation

This specification uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234] with a list extension, defined in Section 7 of [Part1], that allows for compact definition of comma-separated lists using a '#' operator (similar to how the '*' operator indicates repetition). Appendix B describes rules imported from other documents. Appendix C shows the collected grammar with all list

operators expanded to standard ABNF notation.

1.2.1. Delta Seconds

The delta-seconds rule specifies a non-negative integer, representing time in seconds.

`delta-seconds = 1*DIGIT`

A recipient parsing a delta-seconds value and converting it to binary form ought to use an arithmetic type of at least 31 bits of non-negative integer range. If a cache receives a delta-seconds value greater than the greatest integer it can represent, or if any of its subsequent calculations overflows, the cache **MUST** consider the value to be either 2147483648 (2^{31}) or the greatest positive integer it can conveniently represent.

Note: The value 2147483648 is here for historical reasons, effectively represents infinity (over 68 years), and does not need to be stored in binary form; an implementation could produce it as a canned string if any overflow occurs, even if the calculations are performed with an arithmetic type incapable of directly representing that number. What matters here is that an overflow be detected and not treated as a negative value in later calculations.

2. Overview of Cache Operation

Proper cache operation preserves the semantics of HTTP transfers ([Part2]) while eliminating the transfer of information already held in the cache. Although caching is an entirely **OPTIONAL** feature of HTTP, it can be assumed that reusing a cached response is desirable and that such reuse is the default behavior when no requirement or local configuration prevents it. Therefore, HTTP cache requirements are focused on preventing a cache from either storing a non-reusable response or reusing a stored response inappropriately, rather than mandating that caches always store and reuse particular responses.

Each cache entry consists of a cache key and one or more HTTP responses corresponding to prior requests that used the same key. The most common form of cache entry is a successful result of a retrieval request: i.e., a 200 (OK) response to a GET request, which contains a representation of the resource identified by the request target (Section 4.3.1 of [Part2]). However, it is also possible to cache permanent redirects, negative results (e.g., 404 (Not Found)), incomplete results (e.g., 206 (Partial Content)), and responses to methods other than GET if the method's definition allows such caching and defines something suitable for use as a cache key.

The primary cache key consists of the request method and target URI. However, since HTTP caches in common use today are typically limited to caching responses to GET, many caches simply decline other methods and use only the URI as the primary cache key.

If a request target is subject to content negotiation, its cache entry might consist of multiple stored responses, each differentiated by a secondary key for the values of the original request's selecting header fields (Section 4.1).

3. Storing Responses in Caches

A cache **MUST NOT** store a response to any request, unless:

- o The request method is understood by the cache and defined as being cacheable, and
- o the response status code is understood by the cache, and
- o the "no-store" cache directive (see Section 5.2) does not appear in request or response header fields, and
- o the "private" response directive (see Section 5.2.2.6) does not appear in the response, if the cache is shared, and
- o the Authorization header field (see Section 4.2 of [Part7]) does not appear in the request, if the cache is shared, unless the response explicitly allows it (see Section 3.2), and
- o the response either:
 - * contains an Expires header field (see Section 5.3), or
 - * contains a max-age response directive (see Section 5.2.2.8), or
 - * contains a s-maxage response directive (see Section 5.2.2.9) and the cache is shared, or
 - * contains a Cache Control Extension (see Section 5.2.3) that allows it to be cached, or
 - * has a status code that is defined as cacheable by default (see Section 4.2.2), or
 - * contains a public response directive (see Section 5.2.2.5).

Note that any of the requirements listed above can be overridden by a cache-control extension; see Section 5.2.3.

In this context, a cache has "understood" a request method or a response status code if it recognizes it and implements all specified caching-related behavior.

Note that, in normal operation, some caches will not store a response that has neither a cache validator nor an explicit expiration time, as such responses are not usually useful to store. However, caches are not prohibited from storing such responses.

3.1. Storing Incomplete Responses

A response message is considered complete when all of the octets indicated by the message framing ([Part1]) are received prior to the connection being closed. If the request method is GET, the response status code is 200 (OK), and the entire response header section has been received, a cache MAY store an incomplete response message body if the cache entry is recorded as incomplete. Likewise, a 206 (Partial Content) response MAY be stored as if it were an incomplete 200 (OK) cache entry. However, a cache MUST NOT store incomplete or partial content responses if it does not support the Range and Content-Range header fields or if it does not understand the range units used in those fields.

A cache MAY complete a stored incomplete response by making a subsequent range request ([Part5]) and combining the successful response with the stored entry, as defined in Section 3.3. A cache MUST NOT use an incomplete response to answer requests unless the response has been made complete or the request is partial and specifies a range that is wholly within the incomplete response. A cache MUST NOT send a partial response to a client without explicitly marking it as such using the 206 (Partial Content) status code.

3.2. Storing Responses to Authenticated Requests

A shared cache MUST NOT use a cached response to a request with an Authorization header field (Section 4.2 of [Part7]) to satisfy any subsequent request unless a cache directive that allows such responses to be stored is present in the response.

In this specification, the following Cache-Control response directives (Section 5.2.2) have such an effect: must-revalidate, public, s-maxage.

Note that cached responses that contain the "must-revalidate" and/or "s-maxage" response directives are not allowed to be served stale (Section 4.2.4) by shared caches. In particular, a response with either "max-age=0, must-revalidate" or "s-maxage=0" cannot be used to satisfy a subsequent request without revalidating it on the origin

server.

3.3. Combining Partial Content

A response might transfer only a partial representation if the connection closed prematurely or if the request used one or more Range specifiers ([Part5]). After several such transfers, a cache might have received several ranges of the same representation. A cache MAY combine these ranges into a single stored response, and reuse that response to satisfy later requests, if they all share the same strong validator and the cache complies with the client requirements in Section 4.3 of [Part5].

When combining the new response with one or more stored responses, a cache MUST:

- o delete any Warning header fields in the stored response with warn-code 1xx (see Section 5.5);
- o retain any Warning header fields in the stored response with warn-code 2xx; and,
- o use other header fields provided in the new response, aside from Content-Range, to replace all instances of the corresponding header fields in the stored response.

4. Constructing Responses from Caches

When presented with a request, a cache MUST NOT reuse a stored response, unless:

- o The presented effective request URI (Section 5.5 of [Part1]) and that of the stored response match, and
- o the request method associated with the stored response allows it to be used for the presented request, and
- o selecting header fields nominated by the stored response (if any) match those presented (see Section 4.1), and
- o the presented request does not contain the no-cache pragma (Section 5.4), nor the no-cache cache directive (Section 5.2.1), unless the stored response is successfully validated (Section 4.3), and
- o the stored response does not contain the no-cache cache directive (Section 5.2.2.2), unless it is successfully validated (Section 4.3), and

- o the stored response is either:
 - * fresh (see Section 4.2), or
 - * allowed to be served stale (see Section 4.2.4), or
 - * successfully validated (see Section 4.3).

Note that any of the requirements listed above can be overridden by a cache-control extension; see Section 5.2.3.

When a stored response is used to satisfy a request without validation, a cache MUST generate an Age header field (Section 5.1), replacing any present in the response with a value equal to the stored response's `current_age`; see Section 4.2.3.

A cache MUST write through requests with methods that are unsafe (Section 4.2.1 of [Part2]) to the origin server; i.e., a cache is not allowed to generate a reply to such a request before having forwarded the request and having received a corresponding response.

Also, note that unsafe requests might invalidate already stored responses; see Section 4.4.

When more than one suitable response is stored, a cache MUST use the most recent response (as determined by the Date header field). It can also forward the request with "Cache-Control: max-age=0" or "Cache-Control: no-cache" to disambiguate which response to use.

A cache that does not have a clock available MUST NOT use stored responses without revalidating them upon every use.

4.1. Calculating Secondary Keys with Vary

When a cache receives a request that can be satisfied by a stored response that has a Vary header field (Section 7.1.4 of [Part2]), it MUST NOT use that response unless all of the selecting header fields nominated by the Vary header field match in both the original request (i.e., that associated with the stored response), and the presented request.

The selecting header fields from two requests are defined to match if and only if those in the first request can be transformed to those in the second request by applying any of the following:

- o adding or removing whitespace, where allowed in the header field's syntax

- o combining multiple header fields with the same field name (see Section 3.2 of [Part1])
- o normalizing both header field values in a way that is known to have identical semantics, according to the header field's specification (e.g., re-ordering field values when order is not significant; case-normalization, where values are defined to be case-insensitive)

If (after any normalization that might take place) a header field is absent from a request, it can only match another request if it is also absent there.

A Vary header field-value of "*" always fails to match.

The stored response with matching selecting header fields is known as the selected response.

If multiple selected responses are available (potentially including responses without a Vary header field), the cache will need to choose one to use. When a selecting header field has a known mechanism for doing so (e.g., qvalues on Accept and similar request header fields), that mechanism MAY be used to select preferred responses; of the remainder, the most recent response (as determined by the Date header field) is used, as per Section 4.

If no selected response is available, the cache cannot satisfy the presented request. Typically, it is forwarded to the origin server in a (possibly conditional; see Section 4.3) request.

4.2. Freshness

A fresh response is one whose age has not yet exceeded its freshness lifetime. Conversely, a stale response is one where it has.

A response's freshness lifetime is the length of time between its generation by the origin server and its expiration time. An explicit expiration time is the time at which the origin server intends that a stored response can no longer be used by a cache without further validation, whereas a heuristic expiration time is assigned by a cache when no explicit expiration time is available.

A response's age is the time that has passed since it was generated by, or successfully validated with, the origin server.

When a response is "fresh" in the cache, it can be used to satisfy subsequent requests without contacting the origin server, thereby improving efficiency.

The primary mechanism for determining freshness is for an origin server to provide an explicit expiration time in the future, using either the Expires header field (Section 5.3) or the max-age response directive (Section 5.2.2.8). Generally, origin servers will assign future explicit expiration times to responses in the belief that the representation is not likely to change in a semantically significant way before the expiration time is reached.

If an origin server wishes to force a cache to validate every request, it can assign an explicit expiration time in the past to indicate that the response is already stale. Compliant caches will normally validate a stale cached response before reusing it for subsequent requests (see Section 4.2.4).

Since origin servers do not always provide explicit expiration times, caches are also allowed to use a heuristic to determine an expiration time under certain circumstances (see Section 4.2.2).

The calculation to determine if a response is fresh is:

$$\text{response_is_fresh} = (\text{freshness_lifetime} > \text{current_age})$$

freshness_lifetime is defined in Section 4.2.1; current_age is defined in Section 4.2.3.

Clients can send the max-age or min-fresh cache directives in a request to constrain or relax freshness calculations for the corresponding response (Section 5.2.1).

When calculating freshness, to avoid common problems in date parsing:

- o Although all date formats are specified to be case-sensitive, a cache recipient SHOULD match day, week, and timezone names case-insensitively.
- o If a cache recipient's internal implementation of time has less resolution than the value of an HTTP-date, the recipient MUST internally represent a parsed Expires date as the nearest time equal to or earlier than the received value.
- o A cache recipient MUST NOT allow local time zones to influence the calculation or comparison of an age or expiration time.
- o A cache recipient SHOULD consider a date with a zone abbreviation other than GMT or UTC to be invalid for calculating expiration.

Note that freshness applies only to cache operation; it cannot be used to force a user agent to refresh its display or reload a

resource. See Section 6 for an explanation of the difference between caches and history mechanisms.

4.2.1. Calculating Freshness Lifetime

A cache can calculate the freshness lifetime (denoted as `freshness_lifetime`) of a response by using the first match of:

- o If the cache is shared and the `s-maxage` response directive (Section 5.2.2.9) is present, use its value, or
- o If the `max-age` response directive (Section 5.2.2.8) is present, use its value, or
- o If the `Expires` response header field (Section 5.3) is present, use its value minus the value of the `Date` response header field, or
- o Otherwise, no explicit expiration time is present in the response. A heuristic freshness lifetime might be applicable; see Section 4.2.2.

Note that this calculation is not vulnerable to clock skew, since all of the information comes from the origin server.

When there is more than one value present for a given directive (e.g., two `Expires` header fields, multiple `Cache-Control: max-age` directives), the directive's value is considered invalid. Caches are encouraged to consider responses that have invalid freshness information to be stale.

4.2.2. Calculating Heuristic Freshness

Since origin servers do not always provide explicit expiration times, a cache MAY assign a heuristic expiration time when an explicit time is not specified, employing algorithms that use other header field values (such as the `Last-Modified` time) to estimate a plausible expiration time. This specification does not provide specific algorithms, but does impose worst-case constraints on their results.

A cache MUST NOT use heuristics to determine freshness when an explicit expiration time is present in the stored response. Because of the requirements in Section 3, this means that, effectively, heuristics can only be used on responses without explicit freshness whose status codes are defined as cacheable by default (see Section 6.1 of [Part2]), and those responses without explicit freshness that have been marked as explicitly cacheable (e.g., with a `"public"` response directive).

If the response has a Last-Modified header field (Section 2.2 of [Part4]), caches are encouraged to use a heuristic expiration value that is no more than some fraction of the interval since that time. A typical setting of this fraction might be 10%.

When a heuristic is used to calculate freshness lifetime, a cache SHOULD generate a Warning header field with a 113 warn-code (see Section 5.5.4) in the response if its current_age is more than 24 hours and such a warning is not already present.

Note: Section 13.9 of [RFC2616] prohibited caches from calculating heuristic freshness for URIs with query components (i.e., those containing '?'). In practice, this has not been widely implemented. Therefore, origin servers are encouraged to send explicit directives (e.g., Cache-Control: no-cache) if they wish to preclude caching.

4.2.3. Calculating Age

The Age header field is used to convey an estimated age of the response message when obtained from a cache. The Age field value is the cache's estimate of the number of seconds since the response was generated or validated by the origin server. In essence, the Age value is the sum of the time that the response has been resident in each of the caches along the path from the origin server, plus the amount of time it has been in transit along network paths.

The following data is used for the age calculation:

age_value

The term "age_value" denotes the value of the Age header field (Section 5.1), in a form appropriate for arithmetic operation; or 0, if not available.

date_value

The term "date_value" denotes the value of the Date header field, in a form appropriate for arithmetic operations. See Section 7.1.1.2 of [Part2] for the definition of the Date header field, and for requirements regarding responses without it.

now

The term "now" means "the current value of the clock at the host performing the calculation". A host ought to use NTP ([RFC5905]) or some similar protocol to synchronize its clocks to Coordinated Universal Time.

request_time

The current value of the clock at the host at the time the request resulting in the stored response was made.

response_time

The current value of the clock at the host at the time the response was received.

A response's age can be calculated in two entirely independent ways:

1. the "apparent_age": response_time minus date_value, if the local clock is reasonably well synchronized to the origin server's clock. If the result is negative, the result is replaced by zero.
2. the "corrected_age_value", if all of the caches along the response path implement HTTP/1.1. A cache **MUST** interpret this value relative to the time the request was initiated, not the time that the response was received.

```
apparent_age = max(0, response_time - date_value);
```

```
response_delay = response_time - request_time;  
corrected_age_value = age_value + response_delay;
```

These are combined as

```
corrected_initial_age = max(apparent_age, corrected_age_value);
```

unless the cache is confident in the value of the Age header field (e.g., because there are no HTTP/1.0 hops in the Via header field), in which case the corrected_age_value **MAY** be used as the corrected_initial_age.

The current_age of a stored response can then be calculated by adding the amount of time (in seconds) since the stored response was last validated by the origin server to the corrected_initial_age.

```
resident_time = now - response_time;  
current_age = corrected_initial_age + resident_time;
```

4.2.4. Serving Stale Responses

A "stale" response is one that either has explicit expiry information or is allowed to have heuristic expiry calculated, but is not fresh according to the calculations in Section 4.2.

A cache MUST NOT generate a stale response if it is prohibited by an explicit in-protocol directive (e.g., by a "no-store" or "no-cache" cache directive, a "must-revalidate" cache-response-directive, or an applicable "s-maxage" or "proxy-revalidate" cache-response-directive; see Section 5.2.2).

A cache MUST NOT send stale responses unless it is disconnected (i.e., it cannot contact the origin server or otherwise find a forward path) or doing so is explicitly allowed (e.g., by the max-stale request directive; see Section 5.2.1).

A cache SHOULD generate a Warning header field with the 110 warn-code (see Section 5.5.1) in stale responses. Likewise, a cache SHOULD generate a 112 warn-code (see Section 5.5.3) in stale responses if the cache is disconnected.

A cache SHOULD NOT generate a new Warning header field when forwarding a response that does not have an Age header field, even if the response is already stale. A cache need not validate a response that merely became stale in transit.

4.3. Validation

When a cache has one or more stored responses for a requested URI, but cannot serve any of them (e.g., because they are not fresh, or one cannot be selected; see Section 4.1), it can use the conditional request mechanism [Part4] in the forwarded request to give the next inbound server an opportunity to select a valid stored response to use, updating the stored metadata in the process, or to replace the stored response(s) with a new response. This process is known as "validating" or "revalidating" the stored response.

4.3.1. Sending a Validation Request

When sending a conditional request for cache validation, a cache sends one or more precondition header fields containing validator metadata from its stored response(s), which is then compared by recipients to determine whether a stored response is equivalent to a current representation of the resource.

One such validator is the timestamp given in a Last-Modified header field (Section 2.2 of [Part4]), which can be used in an If-Modified-

Since header field for response validation, or in an If-Unmodified-Since or If-Range header field for representation selection (i.e., the client is referring specifically to a previously obtained representation with that timestamp).

Another validator is the entity-tag given in an ETag header field (Section 2.3 of [Part4]). One or more entity-tags, indicating one or more stored responses, can be used in an If-None-Match header field for response validation, or in an If-Match or If-Range header field for representation selection (i.e., the client is referring specifically to one or more previously obtained representations with the listed entity-tags).

4.3.2. Handling a Received Validation Request

Each client in the request chain may have its own cache, so it is common for a cache at an intermediary to receive conditional requests from other (outbound) caches. Likewise, some user agents make use of conditional requests to limit data transfers to recently modified representations or to complete the transfer of a partially retrieved representation.

If a cache receives a request that can be satisfied by reusing one of its stored 200 (OK) or 206 (Partial Content) responses, the cache SHOULD evaluate any applicable conditional header field preconditions received in that request with respect to the corresponding validators contained within the selected response. A cache MUST NOT evaluate conditional header fields that are only applicable to an origin server, found in a request with semantics that cannot be satisfied with a cached response, or applied to a target resource for which it has no stored responses; such preconditions are likely intended for some other (inbound) server.

The proper evaluation of conditional requests by a cache depends on the received precondition header fields and their precedence, as defined in Section 6 of [Part4]. The If-Match and If-Unmodified-Since conditional header fields are not applicable to a cache.

A request containing an If-None-Match header field (Section 3.2 of [Part4]) indicates that the client wants to validate one or more of its own stored responses in comparison to whichever stored response is selected by the cache. If the field-value is "*", or if the field-value is a list of entity-tags and at least one of them match the entity-tag of the selected stored response, a cache recipient SHOULD generate a 304 (Not Modified) response (using the metadata of the selected stored response) instead of sending that stored response.

When a cache decides to revalidate its own stored responses for a request that contains an If-None-Match list of entity-tags, the cache MAY combine the received list with a list of entity-tags from its own stored set of responses (fresh or stale) and send the union of the two lists as a replacement If-None-Match header field value in the forwarded request. If a stored response contains only partial content, the cache MUST NOT include its entity-tag in the union unless the request is for a range that would be fully satisfied by that partial stored response. If the response to the forwarded request is 304 (Not Modified) and has an ETag header field value with an entity-tag that is not in the client's list, the cache MUST generate a 200 (OK) response for the client by reusing its corresponding stored response, as updated by the 304 response metadata (Section 4.3.4).

If an If-None-Match header field is not present, a request containing an If-Modified-Since header field (Section 3.3 of [Part4]) indicates that the client wants to validate one or more of its own stored responses by modification date. A cache recipient SHOULD generate a 304 (Not Modified) response (using the metadata of the selected stored response) if one of the following cases is true: 1) the selected stored response has a Last-Modified field-value that is earlier than or equal to the conditional timestamp; 2) no Last-Modified field is present in the selected stored response, but it has a Date field-value that is earlier than or equal to the conditional timestamp; or, 3) neither Last-Modified nor Date is present in the selected stored response, but the cache recorded it as having been received at a time earlier than or equal to the conditional timestamp.

A cache that implements partial responses to range requests, as defined in [Part5], also needs to evaluate a received If-Range header field (Section 3.2 of [Part5]) with respect to its selected stored response.

4.3.3. Handling a Validation Response

Cache handling of a response to a conditional request is dependent upon its status code:

- o A 304 (Not Modified) response status code indicates that the stored response can be updated and reused; see Section 4.3.4.
- o A full response (i.e., one with a payload body) indicates that none of the stored responses nominated in the conditional request is suitable. Instead, the cache MUST use the full response to satisfy the request and MAY replace the stored response(s).

- o However, if a cache receives a 5xx (Server Error) response while attempting to validate a response, it can either forward this response to the requesting client, or act as if the server failed to respond. In the latter case, the cache MAY send a previously stored response (see Section 4.2.4).

4.3.4. Freshening Stored Responses upon Validation

When a cache receives a 304 (Not Modified) response and already has one or more stored 200 (OK) responses for the same cache key, the cache needs to identify which of the stored responses are updated by this new response and then update the stored response(s) with the new information provided in the 304 response.

The stored response to update is identified by using the first match (if any) of:

- o If the new response contains a strong validator (see Section 2.1 of [Part4]), then that strong validator identifies the selected representation for update. All of the stored responses with the same strong validator are selected. If none of the stored responses contain the same strong validator, then the cache MUST NOT use the new response to update any stored responses.
- o If the new response contains a weak validator and that validator corresponds to one of the cache's stored responses, then the most recent of those matching stored responses is selected for update.
- o If the new response does not include any form of validator (such as in the case where a client generates an If-Modified-Since request from a source other than the Last-Modified response header field), and there is only one stored response, and that stored response also lacks a validator, then that stored response is selected for update.

If a stored response is selected for update, the cache MUST:

- o delete any Warning header fields in the stored response with warn-code 1xx (see Section 5.5);
- o retain any Warning header fields in the stored response with warn-code 2xx; and,
- o use other header fields provided in the 304 (Not Modified) response to replace all instances of the corresponding header fields in the stored response.

4.3.5. Freshening Responses via HEAD

A response to the HEAD method is identical to what an equivalent request made with a GET would have been, except it lacks a body. This property of HEAD responses can be used to invalidate or update a cached GET response if the more efficient conditional GET request mechanism is not available (due to no validators being present in the stored response) or if transmission of the representation body is not desired even if it has changed.

When a cache makes an inbound HEAD request for a given request target and receives a 200 (OK) response, the cache SHOULD update or invalidate each of its stored GET responses that could have been selected for that request (see Section 4.1).

For each of the stored responses that could have been selected, if the stored response and HEAD response have matching values for any received validator fields (ETag and Last-Modified) and, if the HEAD response has a Content-Length header field, the value of Content-Length matches that of the stored response, the cache SHOULD update the stored response as described below; otherwise, the cache SHOULD consider the stored response to be stale.

If a cache updates a stored response with the metadata provided in a HEAD response, the cache MUST:

- o delete any Warning header fields in the stored response with warn-code 1xx (see Section 5.5);
- o retain any Warning header fields in the stored response with warn-code 2xx; and,
- o use other header fields provided in the HEAD response to replace all instances of the corresponding header fields in the stored response and append new header fields to the stored response's header section unless otherwise restricted by the Cache-Control header field.

4.4. Invalidation

Because unsafe request methods (Section 4.2.1 of [Part2]) such as PUT, POST or DELETE have the potential for changing state on the origin server, intervening caches can use them to keep their contents up-to-date.

A cache MUST invalidate the effective Request URI (Section 5.5 of [Part1]) as well as the URI(s) in the Location and Content-Location response header fields (if present) when a non-error status code is

received in response to an unsafe request method.

However, a cache MUST NOT invalidate a URI from a Location or Content-Location response header field if the host part of that URI differs from the host part in the effective request URI (Section 5.5 of [Part1]). This helps prevent denial of service attacks.

A cache MUST invalidate the effective request URI (Section 5.5 of [Part1]) when it receives a non-error response to a request with a method whose safety is unknown.

Here, a "non-error response" is one with a 2xx (Successful) or 3xx (Redirection) status code. "Invalidate" means that the cache will either remove all stored responses related to the effective request URI, or will mark these as "invalid" and in need of a mandatory validation before they can be sent in response to a subsequent request.

Note that this does not guarantee that all appropriate responses are invalidated. For example, a state-changing request might invalidate responses in the caches it travels through, but relevant responses still might be stored in other caches that it has not.

5. Header Field Definitions

This section defines the syntax and semantics of HTTP/1.1 header fields related to caching.

5.1. Age

The "Age" header field conveys the sender's estimate of the amount of time since the response was generated or successfully validated at the origin server. Age values are calculated as specified in Section 4.2.3.

Age = delta-seconds

The Age field-value is a non-negative integer, representing time in seconds (see Section 1.2.1).

The presence of an Age header field implies that the response was not generated or validated by the origin server for this request. However, lack of an Age header field does not imply the origin was contacted, since the response might have been received from an HTTP/1.0 cache that does not implement Age.

5.2. Cache-Control

The "Cache-Control" header field is used to specify directives for caches along the request/response chain. Such cache directives are unidirectional in that the presence of a directive in a request does not imply that the same directive is to be given in the response.

A cache **MUST** obey the requirements of the Cache-Control directives defined in this section. See Section 5.2.3 for information about how Cache-Control directives defined elsewhere are handled.

Note: Some HTTP/1.0 caches might not implement Cache-Control.

A proxy, whether or not it implements a cache, **MUST** pass cache directives through in forwarded messages, regardless of their significance to that application, since the directives might be applicable to all recipients along the request/response chain. It is not possible to target a directive to a specific cache.

Cache directives are identified by a token, to be compared case-insensitively, and have an optional argument, that can use both token and quoted-string syntax. For the directives defined below that define arguments, recipients ought to accept both forms, even if one is documented to be preferred. For any directive not defined by this specification, a recipient **MUST** accept both forms.

Cache-Control = 1#cache-directive

cache-directive = token ["=" (token / quoted-string)]

For the cache directives defined below, no argument is defined (nor allowed) unless stated otherwise.

5.2.1. Request Cache-Control Directives

5.2.1.1. max-age

Argument syntax:

delta-seconds (see Section 1.2.1)

The "max-age" request directive indicates that the client is unwilling to accept a response whose age is greater than the specified number of seconds. Unless the max-stale request directive is also present, the client is not willing to accept a stale response.

This directive uses the token form of the argument syntax; e.g.,

'max-age=5', not 'max-age="5"'. A sender SHOULD NOT generate the quoted-string form.

5.2.1.2. max-stale

Argument syntax:

delta-seconds (see Section 1.2.1)

The "max-stale" request directive indicates that the client is willing to accept a response that has exceeded its freshness lifetime. If max-stale is assigned a value, then the client is willing to accept a response that has exceeded its freshness lifetime by no more than the specified number of seconds. If no value is assigned to max-stale, then the client is willing to accept a stale response of any age.

This directive uses the token form of the argument syntax; e.g., 'max-stale=10', not 'max-stale="10"'. A sender SHOULD NOT generate the quoted-string form.

5.2.1.3. min-fresh

Argument syntax:

delta-seconds (see Section 1.2.1)

The "min-fresh" request directive indicates that the client is willing to accept a response whose freshness lifetime is no less than its current age plus the specified time in seconds. That is, the client wants a response that will still be fresh for at least the specified number of seconds.

This directive uses the token form of the argument syntax; e.g., 'min-fresh=20', not 'min-fresh="20"'. A sender SHOULD NOT generate the quoted-string form.

5.2.1.4. no-cache

The "no-cache" request directive indicates that a cache MUST NOT use a stored response to satisfy the request without successful validation on the origin server.

5.2.1.5. no-store

The "no-store" request directive indicates that a cache MUST NOT store any part of either this request or any response to it. This directive applies to both private and shared caches. "MUST NOT

store" in this context means that the cache MUST NOT intentionally store the information in non-volatile storage, and MUST make a best-effort attempt to remove the information from volatile storage as promptly as possible after forwarding it.

This directive is NOT a reliable or sufficient mechanism for ensuring privacy. In particular, malicious or compromised caches might not recognize or obey this directive, and communications networks might be vulnerable to eavesdropping.

Note that if a request containing this directive is satisfied from a cache, the no-store request directive does not apply to the already stored response.

5.2.1.6. no-transform

The "no-transform" request directive indicates that an intermediary (whether or not it implements a cache) MUST NOT transform the payload, as defined in Section 5.7.2 of [Part1].

5.2.1.7. only-if-cached

The "only-if-cached" request directive indicates that the client only wishes to obtain a stored response. If it receives this directive, a cache SHOULD either respond using a stored response that is consistent with the other constraints of the request, or respond with a 504 (Gateway Timeout) status code. If a group of caches is being operated as a unified system with good internal connectivity, a member cache MAY forward such a request within that group of caches.

5.2.2. Response Cache-Control Directives

5.2.2.1. must-revalidate

The "must-revalidate" response directive indicates that once it has become stale, a cache MUST NOT use the response to satisfy subsequent requests without successful validation on the origin server.

The must-revalidate directive is necessary to support reliable operation for certain protocol features. In all circumstances a cache MUST obey the must-revalidate directive; in particular, if a cache cannot reach the origin server for any reason, it MUST generate a 504 (Gateway Timeout) response.

The must-revalidate directive ought to be used by servers if and only if failure to validate a request on the representation could result in incorrect operation, such as a silently unexecuted financial transaction.

5.2.2.2. no-cache

Argument syntax:

#field-name

The "no-cache" response directive indicates that the response **MUST NOT** be used to satisfy a subsequent request without successful validation on the origin server. This allows an origin server to prevent a cache from using it to satisfy a request without contacting it, even by caches that have been configured to send stale responses.

If the no-cache response directive specifies one or more field-names, then a cache **MAY** use the response to satisfy a subsequent request, subject to any other restrictions on caching. However, any header fields in the response that have the field-name(s) listed **MUST NOT** be sent in the response to a subsequent request without successful revalidation with the origin server. This allows an origin server to prevent the re-use of certain header fields in a response, while still allowing caching of the rest of the response.

The field-names given are not limited to the set of header fields defined by this specification. Field names are case-insensitive.

This directive uses the quoted-string form of the argument syntax. A sender **SHOULD NOT** generate the token form (even if quoting appears not to be needed for single-entry lists).

Note: Although it has been back-ported to many implementations, some HTTP/1.0 caches will not recognize or obey this directive. Also, no-cache response directives with field-names are often handled by caches as if an unqualified no-cache directive was received; i.e., the special handling for the qualified form is not widely implemented.

5.2.2.3. no-store

The "no-store" response directive indicates that a cache **MUST NOT** store any part of either the immediate request or response. This directive applies to both private and shared caches. "MUST NOT store" in this context means that the cache **MUST NOT** intentionally store the information in non-volatile storage, and **MUST** make a best-effort attempt to remove the information from volatile storage as promptly as possible after forwarding it.

This directive is **NOT** a reliable or sufficient mechanism for ensuring privacy. In particular, malicious or compromised caches might not recognize or obey this directive, and communications networks might

be vulnerable to eavesdropping.

5.2.2.4. no-transform

The "no-transform" response directive indicates that an intermediary (regardless of whether it implements a cache) MUST NOT transform the payload, as defined in Section 5.7.2 of [Part1].

5.2.2.5. public

The "public" response directive indicates that any cache MAY store the response, even if the response would normally be non-cacheable or cacheable only within a private cache. (See Section 3.2 for additional details related to the use of public in response to a request containing Authorization, and Section 3 for details of how public affects responses that would normally not be stored, due to their status codes not being defined as cacheable by default; see Section 4.2.2.)

5.2.2.6. private

Argument syntax:

#field-name

The "private" response directive indicates that the response message is intended for a single user and MUST NOT be stored by a shared cache. A private cache MAY store the response and reuse it for later requests, even if the response would normally be non-cacheable.

If the private response directive specifies one or more field-names, this requirement is limited to the field-values associated with the listed response header fields. That is, a shared cache MUST NOT store the specified field-name(s), whereas it MAY store the remainder of the response message.

The field-names given are not limited to the set of header fields defined by this specification. Field names are case-insensitive.

This directive uses the quoted-string form of the argument syntax. A sender SHOULD NOT generate the token form (even if quoting appears not to be needed for single-entry lists).

Note: This usage of the word "private" only controls where the response can be stored; it cannot ensure the privacy of the message content. Also, private response directives with field-names are often handled by caches as if an unqualified private directive was received; i.e., the special handling for the qualified form is not

widely implemented.

5.2.2.7. proxy-revalidate

The "proxy-revalidate" response directive has the same meaning as the must-revalidate response directive, except that it does not apply to private caches.

5.2.2.8. max-age

Argument syntax:

delta-seconds (see Section 1.2.1)

The "max-age" response directive indicates that the response is to be considered stale after its age is greater than the specified number of seconds.

This directive uses the token form of the argument syntax; e.g., 'max-age=5', not 'max-age="5"'. A sender SHOULD NOT generate the quoted-string form.

5.2.2.9. s-maxage

Argument syntax:

delta-seconds (see Section 1.2.1)

The "s-maxage" response directive indicates that, in shared caches, the maximum age specified by this directive overrides the maximum age specified by either the max-age directive or the Expires header field. The s-maxage directive also implies the semantics of the proxy-revalidate response directive.

This directive uses the token form of the argument syntax; e.g., 's-maxage=10', not 's-maxage="10"'. A sender SHOULD NOT generate the quoted-string form.

5.2.3. Cache Control Extensions

The Cache-Control header field can be extended through the use of one or more cache-extension tokens, each with an optional value. A cache MUST ignore unrecognized cache directives.

Informational extensions (those that do not require a change in cache behavior) can be added without changing the semantics of other directives.

Behavioral extensions are designed to work by acting as modifiers to the existing base of cache directives. Both the new directive and the old directive are supplied, such that applications that do not understand the new directive will default to the behavior specified by the old directive, and those that understand the new directive will recognize it as modifying the requirements associated with the old directive. In this way, extensions to the existing cache-control directives can be made without breaking deployed caches.

For example, consider a hypothetical new response directive called "community" that acts as a modifier to the private directive: in addition to private caches, any cache that is shared only by members of the named community is allowed to cache the response. An origin server wishing to allow the UCI community to use an otherwise private response in their shared cache(s) could do so by including

```
Cache-Control: private, community="UCI"
```

A cache that recognizes such a community cache-extension could broaden its behavior in accordance with that extension. A cache that does not recognize the community cache-extension would ignore it and adhere to the private directive.

5.3. Expires

The "Expires" header field gives the date/time after which the response is considered stale. See Section 4.2 for further discussion of the freshness model.

The presence of an Expires field does not imply that the original resource will change or cease to exist at, before, or after that time.

The Expires value is an HTTP-date timestamp, as defined in Section 7.1.1.1 of [Part2].

```
Expires = HTTP-date
```

For example

```
Expires: Thu, 01 Dec 1994 16:00:00 GMT
```

A cache recipient MUST interpret invalid date formats, especially the value "0", as representing a time in the past (i.e., "already expired").

If a response includes a Cache-Control field with the max-age directive (Section 5.2.2.8), a recipient MUST ignore the Expires

field. Likewise, if a response includes the s-maxage directive (Section 5.2.2.9), a shared cache recipient MUST ignore the Expires field. In both these cases, the value in Expires is only intended for recipients that have not yet implemented the Cache-Control field.

An origin server without a clock MUST NOT generate an Expires field unless its value represents a fixed time in the past (always expired) or its value has been associated with the resource by a system or user with a reliable clock.

Historically, HTTP required the Expires field-value to be no more than a year in the future. While longer freshness lifetimes are no longer prohibited, extremely large values have been demonstrated to cause problems (e.g., clock overflows due to use of 32-bit integers for time values), and many caches will evict a response far sooner than that.

5.4. Pragma

The "Pragma" header field allows backwards compatibility with HTTP/1.0 caches, so that clients can specify a "no-cache" request that they will understand (as Cache-Control was not defined until HTTP/1.1). When the Cache-Control header field is also present and understood in a request, Pragma is ignored.

In HTTP/1.0, Pragma was defined as an extensible field for implementation-specified directives for recipients. This specification deprecates such extensions to improve interoperability.

```
Pragma           = 1#pragma-directive
pragma-directive = "no-cache" / extension-pragma
extension-pragma = token [ "=" ( token / quoted-string ) ]
```

When the Cache-Control header field is not present in a request, caches MUST consider the no-cache request pragma-directive as having the same effect as if "Cache-Control: no-cache" were present (see Section 5.2.1).

When sending a no-cache request, a client ought to include both the pragma and cache-control directives, unless Cache-Control: no-cache is purposefully omitted to target other Cache-Control response directives at HTTP/1.1 caches. For example:

```
GET / HTTP/1.1
Host: www.example.com
Cache-Control: max-age=30
Pragma: no-cache
```

will constrain HTTP/1.1 caches to serve a response no older than 30 seconds, while precluding implementations that do not understand Cache-Control from serving a cached response.

Note: Because the meaning of "Pragma: no-cache" in responses is not specified, it does not provide a reliable replacement for "Cache-Control: no-cache" in them.

5.5. Warning

The "Warning" header field is used to carry additional information about the status or transformation of a message that might not be reflected in the status code. This information is typically used to warn about possible incorrectness introduced by caching operations or transformations applied to the payload of the message.

Warnings can be used for other purposes, both cache-related and otherwise. The use of a warning, rather than an error status code, distinguishes these responses from true failures.

Warning header fields can in general be applied to any message, however some warn-codes are specific to caches and can only be applied to response messages.

```
Warning           = 1#warning-value

warning-value = warn-code SP warn-agent SP warn-text
               [ SP warn-date ]

warn-code  = 3DIGIT
warn-agent = ( uri-host [ ":" port ] ) / pseudonym
              ; the name or pseudonym of the server adding
              ; the Warning header field, for use in debugging
              ; a single "-" is recommended when agent unknown
warn-text  = quoted-string
warn-date  = DQUOTE HTTP-date DQUOTE
```

Multiple warnings can be generated in a response (either by the origin server or by a cache), including multiple warnings with the same warn-code number that only differ in warn-text.

A user agent that receives one or more Warning header fields SHOULD inform the user of as many of them as possible, in the order that they appear in the response. Senders that generate multiple Warning header fields are encouraged to order them with this user agent behavior in mind. A sender that generates new Warning header fields MUST append them after any existing Warning header fields.

Warnings are assigned three digit warn-codes. The first digit indicates whether the Warning is required to be deleted from a stored response after validation:

- o 1xx warn-codes describe the freshness or validation status of the response, and so MUST be deleted by a cache after validation. They can only be generated by a cache when validating a cached entry, and MUST NOT be generated in any other situation.
- o 2xx warn-codes describe some aspect of the representation that is not rectified by a validation (for example, a lossy compression of the representation) and MUST NOT be deleted by a cache after validation, unless a full response is sent, in which case they MUST be.

If a sender generates one or more 1xx warn-codes in a message to be sent to a recipient known to implement only HTTP/1.0, the sender MUST include in each corresponding warning-value a warn-date that matches the Date header field in the message. For example:

```
HTTP/1.1 200 OK
Date: Sat, 25 Aug 2012 23:34:45 GMT
Warning: 112 - "network down" "Sat, 25 Aug 2012 23:34:45 GMT"
```

Warnings have accompanying warn-text that describes the error, e.g., for logging. It is advisory only, and its content does not affect interpretation of the warn-code.

If a recipient that uses, evaluates, or displays Warning header fields receives a warn-date that is different from the Date value in the same message, the recipient MUST exclude the warning-value containing that warn-date before storing, forwarding, or using the message. This allows recipients to exclude warning-values that were improperly retained after a cache validation. If all of the warning-values are excluded, the recipient MUST exclude the Warning header field as well.

The following warn-codes are defined by this specification, each with a recommended warn-text in English, and a description of its meaning. The procedure for defining additional warn codes is described in Section 7.2.1.

5.5.1. Warning: 110 - "Response is Stale"

A cache SHOULD generate this whenever the sent response is stale.

5.5.2. Warning: 111 - "Revalidation Failed"

A cache SHOULD generate this when sending a stale response because an attempt to validate the response failed, due to an inability to reach the server.

5.5.3. Warning: 112 - "Disconnected Operation"

A cache SHOULD generate this if it is intentionally disconnected from the rest of the network for a period of time.

5.5.4. Warning: 113 - "Heuristic Expiration"

A cache SHOULD generate this if it heuristically chose a freshness lifetime greater than 24 hours and the response's age is greater than 24 hours.

5.5.5. Warning: 199 - "Miscellaneous Warning"

The warning text can include arbitrary information to be presented to a human user, or logged. A system receiving this warning MUST NOT take any automated action, besides presenting the warning to the user.

5.5.6. Warning: 214 - "Transformation Applied"

MUST be added by a proxy if it applies any transformation to the representation, such as changing the content-coding, media-type, or modifying the representation data, unless this Warning code already appears in the response.

5.5.7. Warning: 299 - "Miscellaneous Persistent Warning"

The warning text can include arbitrary information to be presented to a human user, or logged. A system receiving this warning MUST NOT take any automated action.

6. History Lists

User agents often have history mechanisms, such as "Back" buttons and history lists, that can be used to redisplay a representation retrieved earlier in a session.

The freshness model (Section 4.2) does not necessarily apply to history mechanisms. I.e., a history mechanism can display a previous representation even if it has expired.

This does not prohibit the history mechanism from telling the user

that a view might be stale, or from honoring cache directives (e.g., Cache-Control: no-store).

7. IANA Considerations

7.1. Cache Directive Registry

The HTTP Cache Directive Registry defines the name space for the cache directives. It will be created and maintained at (the suggested URI)
<<http://www.iana.org/assignments/http-cache-directives>>.

7.1.1. Procedure

A registration MUST include the following fields:

- o Cache Directive Name
- o Pointer to specification text

Values to be added to this name space require IETF Review (see [RFC5226], Section 4.1).

7.1.2. Considerations for New Cache Control Directives

New extension directives ought to consider defining:

- o What it means for a directive to be specified multiple times,
- o When the directive does not take an argument, what it means when an argument is present,
- o When the directive requires an argument, what it means when it is missing,
- o Whether the directive is specific to requests, responses, or able to be used in either.

See also Section 5.2.3.

7.1.3. Registrations

The HTTP Cache Directive Registry shall be populated with the registrations below:

| Cache Directive | Reference |
|------------------------|----------------------------------|
| max-age | Section 5.2.1.1, Section 5.2.2.8 |
| max-stale | Section 5.2.1.2 |
| min-fresh | Section 5.2.1.3 |
| must-revalidate | Section 5.2.2.1 |
| no-cache | Section 5.2.1.4, Section 5.2.2.2 |
| no-store | Section 5.2.1.5, Section 5.2.2.3 |
| no-transform | Section 5.2.1.6, Section 5.2.2.4 |
| only-if-cached | Section 5.2.1.7 |
| private | Section 5.2.2.6 |
| proxy-revalidate | Section 5.2.2.7 |
| public | Section 5.2.2.5 |
| s-maxage | Section 5.2.2.9 |
| stale-if-error | [RFC5861], Section 4 |
| stale-while-revalidate | [RFC5861], Section 3 |

7.2. Warn Code Registry

The HTTP Warn Code Registry defines the name space for warn codes. It will be created and maintained at (the suggested URI)
<<http://www.iana.org/assignments/http-warn-codes>>.

7.2.1. Procedure

A registration MUST include the following fields:

- o Warn Code (3 digits)
- o Short Description
- o Pointer to specification text

Values to be added to this name space require IETF Review (see [RFC5226], Section 4.1).

7.2.2. Registrations

The HTTP Warn Code Registry shall be populated with the registrations below:

| Warn Code | Short Description | Reference |
|-----------|----------------------------------|---------------|
| 110 | Response is Stale | Section 5.5.1 |
| 111 | Revalidation Failed | Section 5.5.2 |
| 112 | Disconnected Operation | Section 5.5.3 |
| 113 | Heuristic Expiration | Section 5.5.4 |
| 199 | Miscellaneous Warning | Section 5.5.5 |
| 214 | Transformation Applied | Section 5.5.6 |
| 299 | Miscellaneous Persistent Warning | Section 5.5.7 |

7.3. Header Field Registration

HTTP header fields are registered within the Message Header Field Registry maintained at <http://www.iana.org/assignments/message-headers/message-header-index.html>.

This document defines the following HTTP header fields, so their associated registry entries shall be updated according to the permanent registrations below (see [BCP90]):

| Header Field Name | Protocol | Status | Reference |
|-------------------|----------|----------|-------------|
| Age | http | standard | Section 5.1 |
| Cache-Control | http | standard | Section 5.2 |
| Expires | http | standard | Section 5.3 |
| Pragma | http | standard | Section 5.4 |
| Warning | http | standard | Section 5.5 |

The change controller is: "IETF (iesg@ietf.org) - Internet Engineering Task Force".

8. Security Considerations

This section is meant to inform developers, information providers, and users of known security concerns specific to HTTP caching. More general security considerations are addressed in HTTP messaging [Part1] and semantics [Part2].

Caches expose additional potential vulnerabilities, since the contents of the cache represent an attractive target for malicious exploitation. Because cache contents persist after an HTTP request is complete, an attack on the cache can reveal information long after a user believes that the information has been removed from the network. Therefore, cache contents need to be protected as sensitive

information.

In particular, various attacks might be amplified by being stored in a shared cache; such "cache poisoning" attacks use the cache to distribute a malicious payload to many clients, and are especially effective when an attacker can use implementation flaws, elevated privileges, or other techniques to insert such a response into a cache. One common attack vector for cache poisoning is to exploit differences in message parsing on proxies and in user agents; see Section 3.3.3 of [Part1] for the relevant requirements.

Likewise, implementation flaws (as well as misunderstanding of cache operation) might lead to caching of sensitive information (e.g., authentication credentials) that is thought to be private, exposing it to unauthorized parties.

Furthermore, the very use of a cache can bring about privacy concerns. For example, if two users share a cache, and the first one browses to a site, the second may be able to detect that the other has been to that site, because the resources from it load more quickly, thanks to the cache.

Note that the Set-Cookie response header field [RFC6265] does not inhibit caching; a cacheable response with a Set-Cookie header field can be (and often is) used to satisfy subsequent requests to caches. Servers who wish to control caching of these responses are encouraged to emit appropriate Cache-Control response header fields.

9. Acknowledgments

See Section 10 of [Part1].

10. References

10.1. Normative References

- [Part1] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", draft-ietf-httpbis-p1-messaging-26 (work in progress), February 2014.
- [Part2] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", draft-ietf-httpbis-p2-semantics-26 (work in progress), February 2014.
- [Part4] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests",

draft-ietf-httpbis-p4-conditional-26 (work in progress),
February 2014.

[Part5] Fielding, R., Ed., Lafon, Y., Ed., and J. Reschke, Ed.,
"Hypertext Transfer Protocol (HTTP/1.1): Range Requests",
draft-ietf-httpbis-p5-range-26 (work in progress),
February 2014.

[Part7] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer
Protocol (HTTP/1.1): Authentication",
draft-ietf-httpbis-p7-auth-26 (work in progress),
February 2014.

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

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax
Specifications: ABNF", STD 68, RFC 5234, January 2008.

10.2. Informative References

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

[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.

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

[RFC5861] Nottingham, M., "HTTP Cache-Control Extensions for Stale
Content", RFC 5861, April 2010.

[RFC5905] Mills, D., Martin, J., Ed., Burbank, J., and W. Kasch,
"Network Time Protocol Version 4: Protocol and Algorithms
Specification", RFC 5905, June 2010.

[RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265,
April 2011.

Appendix A. Changes from RFC 2616

The specification has been substantially rewritten for clarity.

The conditions under which an authenticated response can be cached

have been clarified. (Section 3.2)

New status codes can now define that caches are allowed to use heuristic freshness with them. Caches are now allowed to calculate heuristic freshness for URIs with query components. (Section 4.2.2)

The algorithm for calculating age is now less conservative. Caches are now required to handle dates with timezones as if they're invalid, because it's not possible to accurately guess. (Section 4.2.3)

The Content-Location response header field is no longer used to determine the appropriate response to use when validating. (Section 4.3)

The algorithm for selecting a cached negotiated response to use has been clarified in several ways. In particular, it now explicitly allows header-specific canonicalization when processing selecting header fields. (Section 4.1)

Requirements regarding denial of service attack avoidance when performing invalidation have been clarified. (Section 4.4)

Cache invalidation only occurs when a successful response is received. (Section 4.4)

Cache directives are explicitly defined to be case-insensitive. Handling of multiple instances of cache directives when only one is expected is now defined. (Section 5.2)

The "no-store" request directive doesn't apply to responses; i.e., a cache can satisfy a request with no-store on it, and does not invalidate it. (Section 5.2.1.5)

The qualified forms of the private and no-cache cache directives are noted to not be widely implemented; e.g., "private=foo" is interpreted by many caches as simply "private". Additionally, the meaning of the qualified form of no-cache has been clarified. (Section 5.2.2)

The "no-cache" response directive's meaning has been clarified. (Section 5.2.2.2)

The one-year limit on Expires header field values has been removed; instead, the reasoning for using a sensible value is given. (Section 5.3)

The Pragma header field is now only defined for backwards

compatibility; future pragmas are deprecated. (Section 5.4)

Some requirements regarding production and processing of the Warning header fields have been relaxed, as it is not widely implemented. Furthermore, the Warning header field no longer uses RFC 2047 encoding, nor allows multiple languages, as these aspects were not implemented. (Section 5.5)

This specification introduces the Cache Directive and Warn Code Registries, and defines considerations for new cache directives. (Section 7.1 and Section 7.2)

Appendix B. Imported ABNF

The following core rules are included by reference, as defined in Appendix B.1 of [RFC5234]: ALPHA (letters), CR (carriage return), CRLF (CR LF), CTL (controls), DIGIT (decimal 0-9), DQUOTE (double quote), HEXDIG (hexadecimal 0-9/A-F/a-f), LF (line feed), OCTET (any 8-bit sequence of data), SP (space), and VCHAR (any visible US-ASCII character).

The rules below are defined in [Part1]:

| | |
|---------------|--|
| OWS | = <OWS, defined in [Part1], Section 3.2.3> |
| field-name | = <field-name, defined in [Part1], Section 3.2> |
| quoted-string | = <quoted-string, defined in [Part1], Section 3.2.6> |
| token | = <token, defined in [Part1], Section 3.2.6> |
| port | = <port, defined in [Part1], Section 2.7> |
| pseudonym | = <pseudonym, defined in [Part1], Section 5.7.1> |
| uri-host | = <uri-host, defined in [Part1], Section 2.7> |

The rules below are defined in other parts:

| | |
|-----------|--|
| HTTP-date | = <HTTP-date, defined in [Part2], Section 7.1.1.1> |
|-----------|--|

Appendix C. Collected ABNF

In the collected ABNF below, list rules are expanded as per Section 1.2 of [Part1].

Age = delta-seconds

Cache-Control = *("," OWS) cache-directive *(OWS "," [OWS
cache-directive])

Expires = HTTP-date

HTTP-date = <HTTP-date, defined in [Part2], Section 7.1.1.1>

OWS = <OWS, defined in [Part1], Section 3.2.3>

Pragma = *("," OWS) pragma-directive *(OWS "," [OWS
pragma-directive])

Warning = *("," OWS) warning-value *(OWS "," [OWS warning-value]
)

cache-directive = token ["=" (token / quoted-string)]

delta-seconds = 1*DIGIT

extension-pragma = token ["=" (token / quoted-string)]

field-name = <field-name, defined in [Part1], Section 3.2>

port = <port, defined in [Part1], Section 2.7>

pragma-directive = "no-cache" / extension-pragma

pseudonym = <pseudonym, defined in [Part1], Section 5.7.1>

quoted-string = <quoted-string, defined in [Part1], Section 3.2.6>

token = <token, defined in [Part1], Section 3.2.6>

uri-host = <uri-host, defined in [Part1], Section 2.7>

warn-agent = (uri-host [":" port]) / pseudonym

warn-code = 3DIGIT

warn-date = DQUOTE HTTP-date DQUOTE

warn-text = quoted-string

warning-value = warn-code SP warn-agent SP warn-text [SP warn-date
]

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

Changes up to the IETF Last Call draft are summarized in <<http://trac.tools.ietf.org/html/draft-ietf-httpbis-p6-cache-24#appendix-D>>.

D.1. Since draft-ietf-httpbis-p6-cache-24

Closed issues:

- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/499>>: "RFC 1305 ref needs to be updated to 5905"
- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/500>>: "dangling reference to cacheable status codes"
- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/512>>: "APPSDIR review of draft-ietf-httpbis-p6-cache-24"

D.2. Since draft-ietf-httpbis-p6-cache-25

Closed issues:

- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/535>>: "IESG ballot on draft-ietf-httpbis-p6-cache-25"
- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/538>>: "add 'stateless' to Abstract"
- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/542>>: "improve introduction of list rule"
- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/549>>: "augment security considerations with pointers to current research"

Index

1

110 (warn-code) 30
111 (warn-code) 31
112 (warn-code) 31
113 (warn-code) 31
199 (warn-code) 31

2

214 (warn-code) 31
299 (warn-code) 31

A

age 10
Age header field 20

C

cache 4

- cache entry 5
- cache key 5
- Cache-Control header field 21
- D
 - Disconnected Operation (warn-text) 31
- E
 - Expires header field 27
 - explicit expiration time 10
- F
 - fresh 10
 - freshness lifetime 10
- G
 - Grammar
 - Age 20
 - Cache-Control 21
 - cache-directive 21
 - delta-seconds 5
 - Expires 27
 - extension-pragma 28
 - Pragma 28
 - pragma-directive 28
 - warn-agent 29
 - warn-code 29
 - warn-date 29
 - warn-text 29
 - Warning 29
 - warning-value 29
- H
 - Heuristic Expiration (warn-text) 31
 - heuristic expiration time 10
- M
 - max-age (cache directive) 21, 26
 - max-stale (cache directive) 22
 - min-fresh (cache directive) 22
 - Miscellaneous Persistent Warning (warn-text) 31
 - Miscellaneous Warning (warn-text) 31
 - must-revalidate (cache directive) 23
- N
 - no-cache (cache directive) 22, 24
 - no-store (cache directive) 22, 24
 - no-transform (cache directive) 23, 25

| | |
|------------------------------------|----|
| O | |
| only-if-cached (cache directive) | 23 |
| P | |
| Pragma header field | 28 |
| private (cache directive) | 25 |
| private cache | 4 |
| proxy-revalidate (cache directive) | 26 |
| public (cache directive) | 25 |
| R | |
| Response is Stale (warn-text) | 30 |
| Revalidation Failed (warn-text) | 31 |
| S | |
| s-maxage (cache directive) | 26 |
| shared cache | 4 |
| stale | 10 |
| strong validator | 18 |
| T | |
| Transformation Applied (warn-text) | 31 |
| V | |
| validator | 15 |
| W | |
| Warning header field | 29 |

Authors' Addresses

Roy T. Fielding (editor)
Adobe Systems Incorporated
345 Park Ave
San Jose, CA 95110
USA

EMail: fielding@gbiv.com
URI: <http://roy.gbiv.com/>

Mark Nottingham (editor)
Akamai

EMail: mnot@mnot.net
URI: <http://www.mnot.net/>

Julian F. Reschke (editor)
greenbytes GmbH
Hafenweg 16
Muenster, NW 48155
Germany

EMail: julian.reschke@greenbytes.de
URI: <http://greenbytes.de/tech/webdav/>

HTTPbis Working Group
Internet-Draft
Obsoletes: 2616 (if approved)
Updates: 2617 (if approved)
Intended status: Standards Track
Expires: August 10, 2014

R. Fielding, Ed.
Adobe
J. Reschke, Ed.
greenbytes
February 6, 2014

Hypertext Transfer Protocol (HTTP/1.1): Authentication
draft-ietf-httpbis-p7-auth-26

Abstract

The Hypertext Transfer Protocol (HTTP) is a stateless application-level protocol for distributed, collaborative, hypermedia information systems. This document defines the HTTP Authentication framework.

Editorial Note (To be removed by RFC Editor)

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

The current issues list is at <http://tools.ietf.org/wg/httpbis/trac/report/3> and related documents (including fancy diffs) can be found at <http://tools.ietf.org/wg/httpbis/>.

The changes in this draft are summarized in Appendix 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 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.

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.

Table of Contents

| | |
|--|----|
| 1. Introduction | 4 |
| 1.1. Conformance and Error Handling | 4 |
| 1.2. Syntax Notation | 4 |
| 2. Access Authentication Framework | 4 |
| 2.1. Challenge and Response | 4 |
| 2.2. Protection Space (Realm) | 6 |
| 3. Status Code Definitions | 7 |
| 3.1. 401 Unauthorized | 7 |
| 3.2. 407 Proxy Authentication Required | 7 |
| 4. Header Field Definitions | 7 |
| 4.1. WWW-Authenticate | 8 |
| 4.2. Authorization | 8 |
| 4.3. Proxy-Authenticate | 9 |
| 4.4. Proxy-Authorization | 9 |
| 5. IANA Considerations | 10 |
| 5.1. Authentication Scheme Registry | 10 |
| 5.1.1. Procedure | 10 |
| 5.1.2. Considerations for New Authentication Schemes | 10 |
| 5.2. Status Code Registration | 12 |
| 5.3. Header Field Registration | 12 |
| 6. Security Considerations | 12 |
| 6.1. Confidentiality of Credentials | 13 |
| 6.2. Authentication Credentials and Idle Clients | 13 |
| 6.3. Protection Spaces | 14 |
| 7. Acknowledgments | 14 |
| 8. References | 14 |
| 8.1. Normative References | 14 |
| 8.2. Informative References | 15 |
| Appendix A. Changes from RFCs 2616 and 2617 | 16 |
| Appendix B. Imported ABNF | 16 |
| Appendix C. Collected ABNF | 16 |
| Appendix D. Change Log (to be removed by RFC Editor before publication) | 17 |
| D.1. Since draft-ietf-httpbis-p7-auth-24 | 17 |
| D.2. Since draft-ietf-httpbis-p7-auth-25 | 18 |
| Index | 18 |

1. Introduction

HTTP provides a general framework for access control and authentication, via an extensible set of challenge-response authentication schemes, which can be used by a server to challenge a client request and by a client to provide authentication information. This document defines HTTP/1.1 authentication in terms of the architecture defined in [Part1], including the general framework previously described in RFC 2617 and the related fields and status codes previously defined in RFC 2616.

The IANA Authentication Scheme Registry (Section 5.1) lists registered authentication schemes and their corresponding specifications, including the "basic" and "digest" authentication schemes previously defined by RFC 2617.

1.1. Conformance and Error Handling

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].

Conformance criteria and considerations regarding error handling are defined in Section 2.5 of [Part1].

1.2. Syntax Notation

This specification uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234] with a list extension, defined in Section 7 of [Part1], that allows for compact definition of comma-separated lists using a '#' operator (similar to how the '*' operator indicates repetition). Appendix B describes rules imported from other documents. Appendix C shows the collected grammar with all list operators expanded to standard ABNF notation.

2. Access Authentication Framework

2.1. Challenge and Response

HTTP provides a simple challenge-response authentication framework that can be used by a server to challenge a client request and by a client to provide authentication information. It uses a case-insensitive token as a means to identify the authentication scheme, followed by additional information necessary for achieving authentication via that scheme. The latter can either be a comma-separated list of parameters or a single sequence of characters capable of holding base64-encoded information.

Authentication parameters are name=value pairs, where the name token is matched case-insensitively, and each parameter name MUST only occur once per challenge.

auth-scheme = token

auth-param = token BWS "=" BWS (token / quoted-string)

token68 = 1*(ALPHA / DIGIT /
 "_" / "." / "_" / "~" / "+" / "/") * "="

The "token68" syntax allows the 66 unreserved URI characters ([RFC3986]), plus a few others, so that it can hold a base64, base64url (URL and filename safe alphabet), base32, or base16 (hex) encoding, with or without padding, but excluding whitespace ([RFC4648]).

A 401 (Unauthorized) response message is used by an origin server to challenge the authorization of a user agent, including a WWW-Authenticate header field containing at least one challenge applicable to the requested resource.

A 407 (Proxy Authentication Required) response message is used by a proxy to challenge the authorization of a client, including a Proxy-Authenticate header field containing at least one challenge applicable to the proxy for the requested resource.

challenge = auth-scheme [1*SP (token68 / #auth-param)]

Note: Many clients fail to parse a challenge that contains an unknown scheme. A workaround for this problem is to list well-supported schemes (such as "basic") first.

A user agent that wishes to authenticate itself with an origin server -- usually, but not necessarily, after receiving a 401 (Unauthorized) -- can do so by including an Authorization header field with the request.

A client that wishes to authenticate itself with a proxy -- usually, but not necessarily, after receiving a 407 (Proxy Authentication Required) -- can do so by including a Proxy-Authorization header field with the request.

Both the Authorization field value and the Proxy-Authorization field value contain the client's credentials for the realm of the resource being requested, based upon a challenge received in a response (possibly at some point in the past). When creating their values, the user agent ought to do so by selecting the challenge with what it

considers to be the most secure auth-scheme that it understands, obtaining credentials from the user as appropriate. Transmission of credentials within header field values implies significant security considerations regarding the confidentiality of the underlying connection, as described in Section 6.1.

```
credentials = auth-scheme [ 1*SP ( token68 / #auth-param ) ]
```

Upon receipt of a request for a protected resource that omits credentials, contains invalid credentials (e.g., a bad password) or partial credentials (e.g., when the authentication scheme requires more than one round trip), an origin server SHOULD send a 401 (Unauthorized) response that contains a WWW-Authenticate header field with at least one (possibly new) challenge applicable to the requested resource.

Likewise, upon receipt of a request that omits proxy credentials or contains invalid or partial proxy credentials, a proxy that requires authentication SHOULD generate a 407 (Proxy Authentication Required) response that contains a Proxy-Authenticate header field with at least one (possibly new) challenge applicable to the proxy.

A server that receives valid credentials which are not adequate to gain access ought to respond with the 403 (Forbidden) status code (Section 6.5.3 of [Part2]).

HTTP does not restrict applications to this simple challenge-response framework for access authentication. Additional mechanisms can be used, such as authentication at the transport level or via message encapsulation, and with additional header fields specifying authentication information. However, such additional mechanisms are not defined by this specification.

2.2. Protection Space (Realm)

The "realm" authentication parameter is reserved for use by authentication schemes that wish to indicate a scope of protection.

A protection space is defined by the canonical root URI (the scheme and authority components of the effective request URI; see Section 5.5 of [Part1]) of the server being accessed, in combination with the realm value if present. These realms allow the protected resources on a server to be partitioned into a set of protection spaces, each with its own authentication scheme and/or authorization database. The realm value is a string, generally assigned by the origin server, which can have additional semantics specific to the authentication scheme. Note that a response can have multiple challenges with the same auth-scheme but different realms.

The protection space determines the domain over which credentials can be automatically applied. If a prior request has been authorized, the user agent MAY reuse the same credentials for all other requests within that protection space for a period of time determined by the authentication scheme, parameters, and/or user preferences (such as a configurable inactivity timeout). Unless specifically allowed by the authentication scheme, a single protection space cannot extend outside the scope of its server.

For historical reasons, a sender MUST only generate the quoted-string syntax. Recipients might have to support both token and quoted-string syntax for maximum interoperability with existing clients that have been accepting both notations for a long time.

3. Status Code Definitions

3.1. 401 Unauthorized

The 401 (Unauthorized) status code indicates that the request has not been applied because it lacks valid authentication credentials for the target resource. The server generating a 401 response MUST send a WWW-Authenticate header field (Section 4.1) containing at least one challenge applicable to the target resource.

If the request included authentication credentials, then the 401 response indicates that authorization has been refused for those credentials. The user agent MAY repeat the request with a new or replaced Authorization header field (Section 4.2). If the 401 response contains the same challenge as the prior response, and the user agent has already attempted authentication at least once, then the user agent SHOULD present the enclosed representation to the user, since it usually contains relevant diagnostic information.

3.2. 407 Proxy Authentication Required

The 407 (Proxy Authentication Required) status code is similar to 401 (Unauthorized), but indicates that the client needs to authenticate itself in order to use a proxy. The proxy MUST send a Proxy-Authenticate header field (Section 4.3) containing a challenge applicable to that proxy for the target resource. The client MAY repeat the request with a new or replaced Proxy-Authorization header field (Section 4.4).

4. Header Field Definitions

This section defines the syntax and semantics of header fields related to the HTTP authentication framework.

4.1. WWW-Authenticate

The "WWW-Authenticate" header field indicates the authentication scheme(s) and parameters applicable to the target resource.

WWW-Authenticate = 1#challenge

A server generating a 401 (Unauthorized) response MUST send a WWW-Authenticate header field containing at least one challenge. A server MAY generate a WWW-Authenticate header field in other response messages to indicate that supplying credentials (or different credentials) might affect the response.

A proxy forwarding a response MUST NOT modify any WWW-Authenticate fields in that response.

User agents are advised to take special care in parsing the field value, as it might contain more than one challenge, and each challenge can contain a comma-separated list of authentication parameters. Furthermore, the header field itself can occur multiple times.

For instance:

WWW-Authenticate: Newauth realm="apps", type=1,
title="Login to \"apps\"", Basic realm="simple"

This header field contains two challenges; one for the "Newauth" scheme with a realm value of "apps", and two additional parameters "type" and "title", and another one for the "Basic" scheme with a realm value of "simple".

Note: The challenge grammar production uses the list syntax as well. Therefore, a sequence of comma, whitespace, and comma can be considered either as applying to the preceding challenge, or to be an empty entry in the list of challenges. In practice, this ambiguity does not affect the semantics of the header field value and thus is harmless.

4.2. Authorization

The "Authorization" header field allows a user agent to authenticate itself with an origin server -- usually, but not necessarily, after receiving a 401 (Unauthorized) response. Its value consists of credentials containing the authentication information of the user agent for the realm of the resource being requested.

Authorization = credentials

If a request is authenticated and a realm specified, the same credentials are presumed to be valid for all other requests within this realm (assuming that the authentication scheme itself does not require otherwise, such as credentials that vary according to a challenge value or using synchronized clocks).

A proxy forwarding a request **MUST NOT** modify any Authorization fields in that request. See Section 3.2 of [Part6] for details of and requirements pertaining to handling of the Authorization field by HTTP caches.

4.3. Proxy-Authenticate

The "Proxy-Authenticate" header field consists of at least one challenge that indicates the authentication scheme(s) and parameters applicable to the proxy for this effective request URI (Section 5.5 of [Part1]). A proxy **MUST** send at least one Proxy-Authenticate header field in each 407 (Proxy Authentication Required) response that it generates.

Proxy-Authenticate = 1#challenge

Unlike WWW-Authenticate, the Proxy-Authenticate header field applies only to the next outbound client on the response chain. This is because only the client that chose a given proxy is likely to have the credentials necessary for authentication. However, when multiple proxies are used within the same administrative domain, such as office and regional caching proxies within a large corporate network, it is common for credentials to be generated by the user agent and passed through the hierarchy until consumed. Hence, in such a configuration, it will appear as if Proxy-Authenticate is being forwarded because each proxy will send the same challenge set.

Note that the parsing considerations for WWW-Authenticate apply to this header field as well; see Section 4.1 for details.

4.4. Proxy-Authorization

The "Proxy-Authorization" header field allows the client to identify itself (or its user) to a proxy that requires authentication. Its value consists of credentials containing the authentication information of the client for the proxy and/or realm of the resource being requested.

Proxy-Authorization = credentials

Unlike Authorization, the Proxy-Authorization header field applies only to the next inbound proxy that demanded authentication using the

Proxy-Authenticate field. When multiple proxies are used in a chain, the Proxy-Authorization header field is consumed by the first inbound proxy that was expecting to receive credentials. A proxy MAY relay the credentials from the client request to the next proxy if that is the mechanism by which the proxies cooperatively authenticate a given request.

5. IANA Considerations

5.1. Authentication Scheme Registry

The HTTP Authentication Scheme Registry defines the name space for the authentication schemes in challenges and credentials. It will be created and maintained at (the suggested URI)
<<http://www.iana.org/assignments/http-authschemes>>.

5.1.1. Procedure

Registrations MUST include the following fields:

- o Authentication Scheme Name
- o Pointer to specification text
- o Notes (optional)

Values to be added to this name space require IETF Review (see [RFC5226], Section 4.1).

5.1.2. Considerations for New Authentication Schemes

There are certain aspects of the HTTP Authentication Framework that put constraints on how new authentication schemes can work:

- o HTTP authentication is presumed to be stateless: all of the information necessary to authenticate a request MUST be provided in the request, rather than be dependent on the server remembering prior requests. Authentication based on, or bound to, the underlying connection is outside the scope of this specification and inherently flawed unless steps are taken to ensure that the connection cannot be used by any party other than the authenticated user (see Section 2.3 of [Part1]).
- o The authentication parameter "realm" is reserved for defining Protection Spaces as defined in Section 2.2. New schemes MUST NOT use it in a way incompatible with that definition.

- o The "token68" notation was introduced for compatibility with existing authentication schemes and can only be used once per challenge or credential. New schemes thus ought to use the "auth-param" syntax instead, because otherwise future extensions will be impossible.
- o The parsing of challenges and credentials is defined by this specification, and cannot be modified by new authentication schemes. When the auth-param syntax is used, all parameters ought to support both token and quoted-string syntax, and syntactical constraints ought to be defined on the field value after parsing (i.e., quoted-string processing). This is necessary so that recipients can use a generic parser that applies to all authentication schemes.

Note: The fact that the value syntax for the "realm" parameter is restricted to quoted-string was a bad design choice not to be repeated for new parameters.

- o Definitions of new schemes ought to define the treatment of unknown extension parameters. In general, a "must-ignore" rule is preferable over "must-understand", because otherwise it will be hard to introduce new parameters in the presence of legacy recipients. Furthermore, it's good to describe the policy for defining new parameters (such as "update the specification", or "use this registry").
- o Authentication schemes need to document whether they are usable in origin-server authentication (i.e., using WWW-Authenticate), and/or proxy authentication (i.e., using Proxy-Authenticate).
- o The credentials carried in an Authorization header field are specific to the User Agent, and therefore have the same effect on HTTP caches as the "private" Cache-Control response directive (Section 5.2.2.6 of [Part6]), within the scope of the request they appear in.

Therefore, new authentication schemes that choose not to carry credentials in the Authorization header field (e.g., using a newly defined header field) will need to explicitly disallow caching, by mandating the use of either Cache-Control request directives (e.g., "no-store", Section 5.2.1.5 of [Part6]) or response directives (e.g., "private").

5.2. Status Code Registration

The HTTP Status Code Registry located at <http://www.iana.org/assignments/http-status-codes> shall be updated with the registrations below:

| Value | Description | Reference |
|-------|-------------------------------|-------------|
| 401 | Unauthorized | Section 3.1 |
| 407 | Proxy Authentication Required | Section 3.2 |

5.3. Header Field Registration

HTTP header fields are registered within the Message Header Field Registry maintained at <http://www.iana.org/assignments/message-headers/message-header-index.html>.

This document defines the following HTTP header fields, so their associated registry entries shall be updated according to the permanent registrations below (see [BCP90]):

| Header Field Name | Protocol | Status | Reference |
|---------------------|----------|----------|-------------|
| Authorization | http | standard | Section 4.2 |
| Proxy-Authenticate | http | standard | Section 4.3 |
| Proxy-Authorization | http | standard | Section 4.4 |
| WWW-Authenticate | http | standard | Section 4.1 |

The change controller is: "IETF (iesg@ietf.org) - Internet Engineering Task Force".

6. Security Considerations

This section is meant to inform developers, information providers, and users of known security concerns specific to HTTP authentication. More general security considerations are addressed in HTTP messaging [Part1] and semantics [Part2].

Everything about the topic of HTTP authentication is a security consideration, so the list of considerations below is not exhaustive. Furthermore, it is limited to security considerations regarding the authentication framework, in general, rather than discussing all of the potential considerations for specific authentication schemes (which ought to be documented in the specifications that define those

schemes). Various organizations maintain topical information and links to current research on Web application security (e.g., [OWASP]), including common pitfalls for implementing and using the authentication schemes found in practice.

6.1. Confidentiality of Credentials

The HTTP authentication framework does not define a single mechanism for maintaining the confidentiality of credentials; instead, each authentication scheme defines how the credentials are encoded prior to transmission. While this provides flexibility for the development of future authentication schemes, it is inadequate for the protection of existing schemes that provide no confidentiality on their own, or that do not sufficiently protect against replay attacks. Furthermore, if the server expects credentials that are specific to each individual user, the exchange of those credentials will have the effect of identifying that user even if the content within credentials remains confidential.

HTTP depends on the security properties of the underlying transport or session-level connection to provide confidential transmission of header fields. In other words, if a server limits access to authenticated users using this framework, the server needs to ensure that the connection is properly secured in accordance with the nature of the authentication scheme used. For example, services that depend on individual user authentication often require a connection to be secured with TLS ("Transport Layer Security", [RFC5246]) prior to exchanging any credentials.

6.2. Authentication Credentials and Idle Clients

Existing HTTP clients and user agents typically retain authentication information indefinitely. HTTP does not provide a mechanism for the origin server to direct clients to discard these cached credentials, since the protocol has no awareness of how credentials are obtained or managed by the user agent. The mechanisms for expiring or revoking credentials can be specified as part of an authentication scheme definition.

Circumstances under which credential caching can interfere with the application's security model include but are not limited to:

- o Clients that have been idle for an extended period, following which the server might wish to cause the client to re-prompt the user for credentials.
- o Applications that include a session termination indication (such as a "logout" or "commit" button on a page) after which the server

side of the application "knows" that there is no further reason for the client to retain the credentials.

User agents that cache credentials are encouraged to provide a readily accessible mechanism for discarding cached credentials under user control.

6.3. Protection Spaces

Authentication schemes that solely rely on the "realm" mechanism for establishing a protection space will expose credentials to all resources on an origin server. Clients that have successfully made authenticated requests with a resource can use the same authentication credentials for other resources on the same origin server. This makes it possible for a different resource to harvest authentication credentials for other resources.

This is of particular concern when an origin server hosts resources for multiple parties under the same canonical root URI (Section 2.2). Possible mitigation strategies include restricting direct access to authentication credentials (i.e., not making the content of the Authorization request header field available), and separating protection spaces by using a different host name (or port number) for each party.

7. Acknowledgments

This specification takes over the definition of the HTTP Authentication Framework, previously defined in RFC 2617. We thank John Franks, Phillip M. Hallam-Baker, Jeffery L. Hostetler, Scott D. Lawrence, Paul J. Leach, Ari Luotonen, and Lawrence C. Stewart for their work on that specification. See Section 6 of [RFC2617] for further acknowledgements.

See Section 10 of [Part1] for the Acknowledgments related to this document revision.

8. References

8.1. Normative References

- [Part1] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", draft-ietf-httpbis-pl-messaging-26 (work in progress), February 2014.
- [Part2] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content",

draft-ietf-httpbis-p2-semantic-26 (work in progress),
February 2014.

- [Part6] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Caching", draft-ietf-httpbis-p6-cache-26 (work in progress), February 2014.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008.

8.2. Informative References

- [BCP90] Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, September 2004.
- [OWASP] van der Stock, A., Ed., "A Guide to Building Secure Web Applications and Web Services", The Open Web Application Security Project (OWASP) 2.0.1, July 2005, <<https://www.owasp.org/>>.
- [RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.
- [RFC2617] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and L. Stewart, "HTTP Authentication: Basic and Digest Access Authentication", RFC 2617, June 1999.
- [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.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

Appendix A. Changes from RFCs 2616 and 2617

The framework for HTTP Authentication is now defined by this document, rather than RFC 2617.

The "realm" parameter is no longer always required on challenges; consequently, the ABNF allows challenges without any auth parameters. (Section 2)

The "token68" alternative to auth-param lists has been added for consistency with legacy authentication schemes such as "Basic". (Section 2)

This specification introduces the Authentication Scheme Registry, along with considerations for new authentication schemes. (Section 5.1)

Appendix B. Imported ABNF

The following core rules are included by reference, as defined in Appendix B.1 of [RFC5234]: ALPHA (letters), CR (carriage return), CRLF (CR LF), CTL (controls), DIGIT (decimal 0-9), DQUOTE (double quote), HEXDIG (hexadecimal 0-9/A-F/a-f), LF (line feed), OCTET (any 8-bit sequence of data), SP (space), and VCHAR (any visible US-ASCII character).

The rules below are defined in [Part1]:

| | |
|---------------|--|
| BWS | = <BWS, defined in [Part1], Section 3.2.3> |
| OWS | = <OWS, defined in [Part1], Section 3.2.3> |
| quoted-string | = <quoted-string, defined in [Part1], Section 3.2.6> |
| token | = <token, defined in [Part1], Section 3.2.6> |

Appendix C. Collected ABNF

In the collected ABNF below, list rules are expanded as per Section 1.2 of [Part1].

```
Authorization = credentials

BWS = <BWS, defined in [Part1], Section 3.2.3>

OWS = <OWS, defined in [Part1], Section 3.2.3>

Proxy-Authenticate = *( "," OWS ) challenge *( OWS "," [ OWS
  challenge ] )
Proxy-Authorization = credentials

WWW-Authenticate = *( "," OWS ) challenge *( OWS "," [ OWS challenge
  ] )

auth-param = token BWS "=" BWS ( token / quoted-string )
auth-scheme = token

challenge = auth-scheme [ 1*SP ( token68 / [ ( "," / auth-param ) *(
  OWS "," [ OWS auth-param ] ) ] ) ]
credentials = auth-scheme [ 1*SP ( token68 / [ ( "," / auth-param )
  *( OWS "," [ OWS auth-param ] ) ] ) ]

quoted-string = <quoted-string, defined in [Part1], Section 3.2.6>

token = <token, defined in [Part1], Section 3.2.6>
token68 = 1*( ALPHA / DIGIT / "-" / "." / "_" / "~" / "+" / "/" )
  * "="
```

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

Changes up to the IETF Last Call draft are summarized in <<http://trac.tools.ietf.org/html/draft-ietf-httpbis-p7-auth-24#appendix-D>>.

D.1. Since draft-ietf-httpbis-p7-auth-24

Closed issues:

- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/510>>: "SECDIR review of draft-ietf-httpbis-p7-auth-24"
- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/513>>: "APPSDIR review of draft-ietf-httpbis-p7-auth-24"
- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/516>>: "note about WWW-A parsing potentially misleading"

D.2. Since draft-ietf-httpbis-p7-auth-25

Closed issues:

- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/522>>: "Gen-art review of draft-ietf-httpbis-p7-auth-25"
- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/536>>: "IESG ballot on draft-ietf-httpbis-p7-auth-25"
- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/538>>: "add 'stateless' to Abstract"
- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/539>>: "mention TLS vs plain text passwords or dict attacks?"
- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/542>>: "improve introduction of list rule"
- o <<http://tools.ietf.org/wg/httpbis/trac/ticket/549>>: "augment security considerations with pointers to current research"

Index

| | |
|---|---|
| 4 | |
| 401 Unauthorized (status code) | 7 |
| 407 Proxy Authentication Required (status code) | 7 |
| A | |
| Authorization header field | 8 |
| C | |
| Canonical Root URI | 6 |
| G | |
| Grammar | |
| auth-param | 5 |
| auth-scheme | 5 |
| Authorization | 8 |
| challenge | 5 |
| credentials | 6 |
| Proxy-Authenticate | 9 |
| Proxy-Authorization | 9 |
| token68 | 5 |
| WWW-Authenticate | 8 |
| P | |
| Protection Space | 6 |

Proxy-Authenticate header field 9
Proxy-Authorization header field 9

R
Realm 6

W
WWW-Authenticate header field 8

Authors' Addresses

Roy T. Fielding (editor)
Adobe Systems Incorporated
345 Park Ave
San Jose, CA 95110
USA

EMail: fielding@gbiv.com
URI: <http://roy.gbiv.com/>

Julian F. Reschke (editor)
greenbytes GmbH
Hafenweg 16
Muenster, NW 48155
Germany

EMail: julian.reschke@greenbytes.de
URI: <http://greenbytes.de/tech/webdav/>

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]):

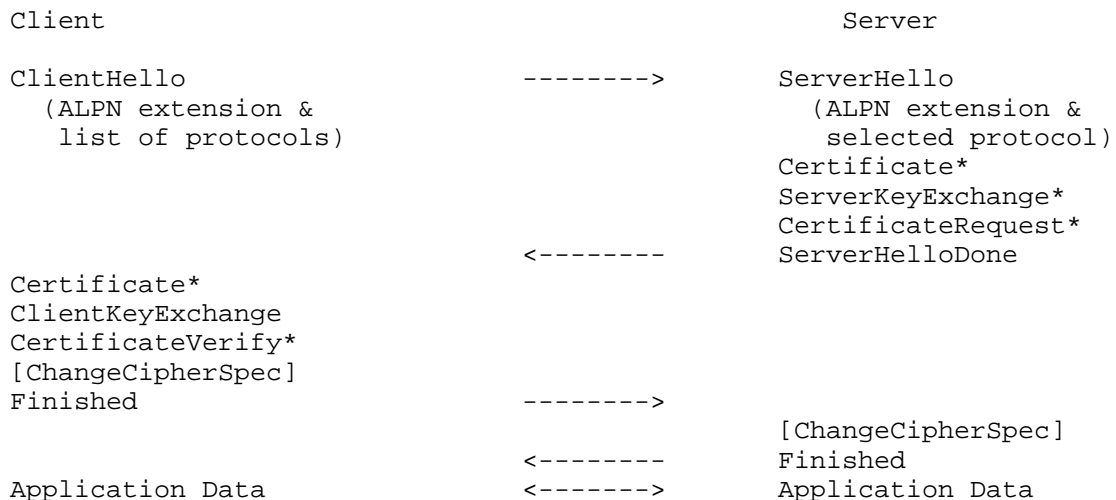


Figure 1

An abbreviated handshake with the "application_layer_protocol_negotiation" extension has the following flow:

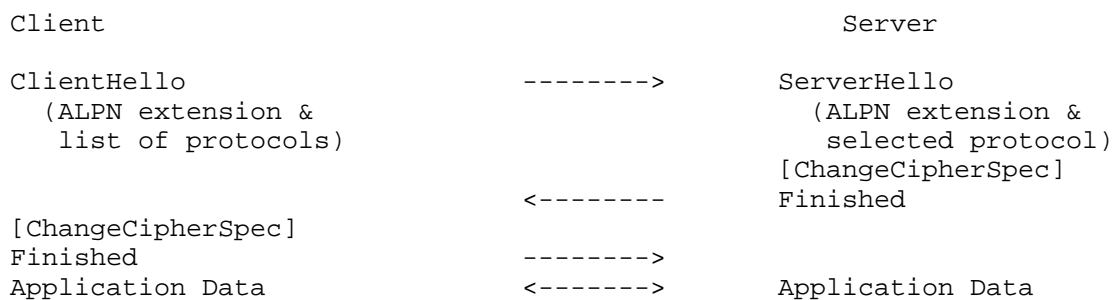


Figure 2

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

HTTPBis Working Group
Internet-Draft
Intended status: Informational
Expires: April 24, 2014

S. Loreto
R. Skog
H. Spaak
Ericsson
D. Druta
M. Hafeez
AT&T
October 21, 2013

2.0 Proxy in HTTP/2.0
draft-loreto-httpbis-proxy20-01

Abstract

This document defines and clarifies the role of proxies (aka intermediaries) in HTTP 2.0. It aims to assure that HTTP 2.0 contains the same proxy features present in HTTP 1.1. It also defines HTTP 2.0 proxies and advocates the importance and the benefits that they can provide for HTTP 2.0. This document aims to start the discussion within the HTTPBis wg.

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 April 24, 2014.

Copyright Notice

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

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents

carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

| | |
|--|---|
| 1. Introduction | 3 |
| 2. Terminology | 4 |
| 3. Current Proxy usages | 4 |
| 4. HTTP 2.0 Proxy | 5 |
| 4.1. HTTP 2.0 Proxy Features | 5 |
| 4.2. 2.0 Proxy: discovery mechanisms | 6 |
| 5. Acknowledgments | 6 |
| 6. References | 6 |
| 6.1. Normative References | 6 |
| 6.2. Informative References | 7 |
| Authors' Addresses | 7 |

1. Introduction

Proxies (aka intermediaries) are an important part of existing HTTP deployments; they both significantly help to support and improve the scalability needs of the Web. Large enterprise deployments are leveraging proxies in their architecture and home networking solutions depend on intermediaries in order to connect to the internet. These are just two examples of the use cases relying on intermediaries and the current deployment scale will demand increased compatibility with existing and future services. However till now little effort has been spent to define and clarify the role intermediaries play in HTTP 2.0.

This draft aims to assure that HTTP/2.0 contains the same proxy features present in HTTP/1.1 [I-D.ietf-httpbis-pl-messaging], [RFC2616] and [RFC2817].

This draft also advocates the importance and the benefits that proxies can provide for HTTP/2.0 and aims to start a discussion on this topic within the HTTPBis wg. Caching is not addressed in this initial version of the document.

A Proxy is defined in HTTP/1.1 is a

"a message forwarding agent that is selected by the client, usually via local configuration rules, to receive requests for some type(s) of absolute URI and attempt to satisfy those requests via translation through the HTTP interface."

A proxy acts as a server and a client for the purpose of making requests on behalf of the client. The requests can be serviced internally (i.e. if the Proxy also implement a cache) or by passing them on to the origin server.

Proxies are often used to group an organization's HTTP requests through a common intermediary for the sake of security, annotation services, or shared caching.

Moreover Proxies can improve the Quality of Experience (QoE) in particular scenarios, such as in a mobile network.

There are important Proxy uses cases currently used in HTTP 1.1 and most likely they will be also important for 2.0 (see Section 3). There are also Proxy use cases that will be 2.0 specific (see Section 4).

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].

2. Terminology

HTTP1.1 [I-D.ietf-httpbis-pl-messaging] defines three form of intermediaries:

Transforming proxy: Is designed and configured to modify request or response messages in a semantically meaningful way. Such transformation is presumed to be desired by the client that selected the proxy.

Gateway (a.k.a., reverse proxy): is an intermediary that acts as an origin server for the outbound connection, but translates received requests and forwards them inbound to another server or servers.

Gateways also include proxies that transform requests from one transport protocol to another, e.g. as in WAP (Wireless Application Protocol) gateways which transformed WAP1 transport to HTTP 1.1. For HTTP 2.0, which includes many of the same features as WAP1 (e.g. header compression, body compression, tunneling of multiple requests over a single connection/session, push, etc), proxies may also need to provide such a transformation, e.g. to enable a single client transport protocol connection (HTTP 2.0), but transform that to an HTTP 1.1 connection for servers that do not support HTTP 2.0.

Tunnel: acts as a blind relay between two connections without changing the messages. Once active, a tunnel is not considered a party to the HTTP communication, though the tunnel might have been initiated by an HTTP request. A tunnel ceases to exist when both ends of the relayed connection are closed. Tunnels are used to extend a virtual connection through an intermediary.

HTTP2.0 defines the following intermediaries:

2.0 Proxy: an interposed entity the user-agent is informed about its existence (by explicit configuration or other TBD mechanisms), and that can be easily bypassed if the user-agent decide to do it.

3. Current Proxy usages

Among all the possible existing Proxy usages, there are some that really improve the user QoE and also help the users while accessing the Web.

Network access control

Protocol Enhancement Proxy

Perform DNS requests on behalf of the user

The list above only enumerates some of the reliable Proxy usage that would provide value also in 2.0.

4. HTTP 2.0 Proxy

In HTTP 2.0 an interposed proxy should always be discoverable by the user-agent so that the user can consent it to stay or easily bypass it.

The actual discovery mechanism is not discussed in this draft.

4.1. HTTP 2.0 Proxy Features

"2.0 Proxy" will provide in additions to the usages listed in Section 4 also other specific HTTP 2.0 usages. Some of them are listed below.

Protocol version translation such as HTTP2.0 vs HTTP1.0/1.1 The HTTP 2.0 benefits will be valuable when it is used in mobile networks; however not all sites on Internet will support 2.0 from day one. The "2.0 Proxy" receives the HTTP 2.0 request by the user-agent and sends a new request to the Origin Server. If the Origin Server does not support HTTP 2.0 then the "2.0 Proxy" will translate the HTTP 2.0 request to HTTP 1.1 request and will then translate the HTTP 1.1 response to the HTTP 2.0 response.

Improve the HTTP/2.0 Flow Control management If both the user-agent and the origin server support HTTP 2.0, the 2.0 proxy can improve the Flow Control management as it is a hop-by-hop mechanism defined to protect endpoints that are operating under resource constraints.

Stream priority management A "2.0 Proxy" can improve the transmission order for streams based on knowledge of the network bandwidth-delay.

Push streams management A "2.0 Proxy" can adhere to network condition and apply push streams management policy. As an example a mobile user roaming can have policy saying not allowed push.

4.2. 2.0 Proxy: discovery mechanisms

The end user, using HTTP 2.0, should become always aware of the existence of any "2.0 Proxy" present in the network. The end user may also be entitled to explicitly consent to use it or to bypass it, although such entitlement may be limited for some applications or service environments.

The actual discovery mechanism is not discussed in this draft. The human factor implications of "proxy awareness" by the user are also not discussed.

5. Acknowledgments

The authors wish to thank Thorsten Herber, Preeyaa Rawlani, Bryan Sullivan for their invaluable comments.

6. References

6.1. Normative References

- [I-D.ietf-httpbis-http2]
Belshe, M., Peon, R., Thomson, M., and A. Melnikov,
"Hypertext Transfer Protocol version 2.0",
draft-ietf-httpbis-http2-06 (work in progress),
August 2013.
- [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-24 (work in progress),
September 2013.
- [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.
- [RFC2817] Khare, R. and S. Lawrence, "Upgrading to TLS Within
HTTP/1.1", RFC 2817, May 2000.
- [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

- [RFC3040] Cooper, I., Melve, I., and G. Tomlinson, "Internet Web Replication and Caching Taxonomy", RFC 3040, January 2001.
- [RFC4732] Handley, M., Rescorla, E., and IAB, "Internet Denial-of-Service Considerations", RFC 4732, December 2006.

Authors' Addresses

Salvatore Loreto
Ericsson
Hirsalantie 11
Jorvas 02420
Finland

Email: salvatore.loreto@ericsson.com

Robert Skog
Ericsson
Sweeden

Email: robert.skog@ericsson.com

Hans Spaak
Ericsson
Sweeden

Email: hans.spaak@ericsson.com

Dan Druta
AT&T

Phone:
Email: dd5826@att.com

Mohammad Hafeez
AT&T

Phone:
Email: mh2897@att.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: January 05, 2015

M. Nottingham
July 04, 2014

Problems with Proxies in HTTP
draft-nottingham-http-proxy-problem-01

Abstract

This document discusses the use and configuration of proxies in HTTP, pointing out problems in the currently deployed Web infrastructure along the way. It then offers a few principles to base further discussion upon, and lists some potential avenues for further exploration.

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 January 05, 2015.

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 |
| 1.1. | Notational Conventions | 3 |
| 2. | Why Proxy? | 3 |
| 2.1. | Application Layer Gatewaying | 4 |
| 2.2. | Caching | 4 |
| 2.3. | Network Policy Enforcement | 4 |
| 2.4. | Content Filtering (a.k.a. Content Policy Enforcement) . . | 4 |
| 2.5. | Content Modification | 5 |
| 3. | How Proxies are Interposed | 6 |
| 3.1. | Manual Configuration | 6 |
| 3.2. | proxy.pac and WPAD | 6 |
| 3.3. | Interception | 7 |
| 3.4. | Configuration As Side Effect | 8 |
| 4. | Second-Order Effects of Proxy Deployment | 8 |
| 4.1. | Proxies and HTTP | 8 |
| 4.2. | Proxies and TLS | 9 |
| 5. | Principles for Consideration | 9 |
| 5.1. | Proxies Have a Legitimate Place | 10 |
| 5.2. | Security Should be Encouraged | 10 |
| 5.3. | Users Need to be Informed of Proxies | 10 |
| 5.4. | Users Need to be able to Tunnel through Proxies | 11 |
| 5.5. | Proxies Can say "No" | 11 |
| 5.6. | Changes Need to be Detectable | 11 |
| 5.7. | Proxies Need to be Easy | 11 |
| 5.8. | Proxies Need to Communicate to Users | 11 |
| 5.9. | Users Require Simple Interfaces | 12 |
| 5.10. | User Agents Are Diverse | 12 |
| 5.11. | RFC2119 Doesn't Define Reality | 12 |
| 5.12. | It Needs to be Deployable | 13 |
| 6. | Potential Areas to Investigate | 13 |
| 6.1. | Improving Proxy.Pac | 13 |
| 6.2. | TLS Errors for Proxies | 13 |
| 6.3. | HTTP Errors for Proxies | 13 |
| 6.4. | TLS for Proxy Connections | 14 |
| 6.5. | Improved Network Information | 14 |
| 6.6. | Improving Trust | 14 |
| 6.7. | HTTP Signatures | 14 |
| 7. | Security Considerations | 15 |
| 8. | Acknowledgements | 15 |
| 9. | References | 15 |
| 9.1. | Normative References | 15 |
| 9.2. | Informative References | 15 |
| | Author's Address | 16 |

1. Introduction

HTTP/1.1 [RFC7230] was designed to accommodate proxies. It allows them (and other components) to cache content expansively, and allows for proxies to break "semantic transparency" by changing message content, within broad constraints.

As the Web has matured, more networks have taken advantage of this by deploying proxies for a variety of reasons, in a number of different ways. Section 2 is a survey of the different ways that proxies are used, and Section 3 shows how they are interposed into communication.

Some uses of proxies cause problems (or the perception of them) for origin servers and end users. While some uses are obviously undesirable from the perspective of an end users and/or origin server, other effects of their deployment are more subtle; these are examined in Section 4.

These tensions between the interests of the stakeholders in every HTTP connection - the end users, the origin servers and the networks they use - has led to decreased trust for proxies, then increasing deployment of encryption, then workarounds for encryption, and so forth.

Left unchecked, this escalation can erode the value of the Web itself. Therefore, Section 5 proposes straw-man principals to base further discussion upon.

Finally, Section 6 proposes some areas of technical investigation that might yield solutions (or at least mitigations) for some of these problems.

Note that this document is explicitly about "proxies" in the sense that HTTP defines them. Intermediaries that are interposed by the server (e.g., gateways and so-called "Reverse Proxies", as used in Content Delivery Networks) are out of scope.

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].

2. Why Proxy?

HTTP proxies are interposed between user agents and origin servers for a variety of purposes; some of them are with the full knowledge and consent of end users, to their benefit, and some are solely for the purposes of the network operator - sometimes even against the interests of the end users.

This section attempts to identify the different motivations networks have for deploying proxies.

2.1. Application Layer Gatewaying

Some networks do not have direct Internet connectivity for Web browsing. These networks can deploy proxies that do have Internet connectivity and then configure clients to use them.

Such gatewaying between networks were some of the first uses for proxies.

2.2. Caching

An extremely common use of proxies is to interpose a HTTP cache, in order to save bandwidth, improve end-user perceived latency, increase reliability, or some combination of these purposes.

HTTP defines a detailed model for caching (see [RFC7234]); however, some lesser-known aspects of the caching model can cause operational issues. For example, it allows caches to go into an "offline" mode where most content can be served stale.

Also, proxy caches sometimes fail to honor the HTTP caching model, reusing content when it should not have been. This can cause interoperability issues, with the end user seeing overly "stale" content, or applications not operating correctly.

2.3. Network Policy Enforcement

Some proxies are deployed to aid in network policy enforcement; for example, to control access to the network, requiring a login (as allowed explicitly by HTTP's proxy authentication mechanism), bandwidth shaping of HTTP access, quotas, etc. This includes so-called "Captive Portals" used for network login.

Some uses of proxies for policy enforcement cause problems; e.g., when a proxy uses URL rewriting to send a user a message (e.g., a "blocked" page), they can make it appear as if the origin server is sending that message - especially when the user agent isn't a browser (e.g., a software update process).

2.4. Content Filtering (a.k.a. Content Policy Enforcement)

Some networks attempt to filter HTTP messages (both request and response) based upon network-specific criteria. For example, they might wish to stop users from downloading content that contains malware, or that violates site policies on appropriate content, or that violates local law.

Intermediary proxies as a mechanism for enforcing content restrictions are often easy to circumvent. For example, a device might become infected by using a different network, or a VPN. Nevertheless, they are commonly used for this purpose.

Some content policy enforcement is also done locally to the user agent; for example, several Operating Systems have machine-local proxies built in that scan content.

Content filtering is often seen as controversial, often depending on the context it is used within and how it is performed.

2.5. Content Modification

Some networks modify HTTP messages (both request and response) as they pass through proxies. This might include the message body, headers, request-target, method or status code.

Motivation for content modification varies. For example, some mobile networks interpose proxies that modify content in an attempt to save bandwidth, improve perceived performance, or transcode content to formats that limited-resource devices can more easily consume.

Modifications also include adding metadata in headers for accounting purposes, or removing metadata such as Accept-Encoding to make virus scanning easier.

In other cases, content modification is performed to make more substantial modifications. This could include inserting advertisements, or changing the layout of content in an attempt to make it easier to use.

Content modification is very controversial, often depending on the context it is used within and how it is performed. Many feel that, without the explicit consent of either the end user or the origin server, a proxy that modifies content violates their relationship, thereby degrading trust in the Web overall.

However, it should be noted that Section 5.7.2 of [RFC7230] explicitly allows "non-transparent" proxies that modify content in certain ways. Such proxies are required to honor the "no-transform" directive, giving both user agents and origin servers a mechanism to

"opt out" of modifications ([RFC7234], Section 5.2.1.6); however, it is not technically enforced.

[W3C.NOTE-ct-guidelines-20101026] is a product of the W3C Mobile Web Best Practices Working Group that attempts to set guidelines for content modification proxies. Again, it is a policy document, without technical enforcement measures.

3. How Proxies are Interposed

How a proxy is interposed into a network flow often has great affect on perceptions of its operation by end users and origin servers. This section catalogues the ways that this happens, and potential problems with each.

3.1. Manual Configuration

The original way to interpose a proxy was to manually configure it into the user agent. For example, most browsers still have the ability to have a proxy hostname and port configured for HTTP; many Operating Systems have system-wide proxy settings.

Unfortunately, manual configuration suffers from several problems:

- o Users often lack the expertise to manually configure proxies.
- o When the user changes networks, they must manually change proxy settings, a laborious task. This makes manual configuration impractical in a modern, mobile-driven world.
- o Not all HTTP stacks support manual proxy configuration. Therefore, a proxy administrator cannot rely upon this method.

3.2. proxy.pac and WPAD

The limitations of manual configuration were recognized long ago. The solution that evolved was a format called "proxy.pac" [proxypac] that allowed the proxy configuration to be automated, once the user agent had loaded it.

Proxy.pac is a JavaScript format; before each request is made, it is dispatched to a function in the file that returns a string that denotes whether a proxy is to be used, and if so, which one to use.

Discovery of the appropriate proxy.pac file for a given network can be made using a DHCP extension, [wpad]. WPAD started as a simple protocol; it conveys a URL that locates the proxy.pac file for the network.

Unfortunately, the proxy.pac/WPAD combination has several operational issues that limit its deployment:

- o The proxy.pac format does not define timeouts or failover behavior precisely, leading to wide divergence between implementations. This makes supporting multiple user agents reliably difficult for the network.
- o WPAD is not widely implemented by user agents; some only implement proxy.pac.
- o In those user agents where it is implemented, WPAD is often not the default, meaning that users need to configure its use.
- o Neither proxy.pac nor WPAD have been standardized, leading to implementation divergence and resulting interoperability problems.
- o There are DNS-based variants of WPAD, adding to to confusion.
- o DHCP options generally require tight integration with the operating system to pass the results to HTTP-based applications. While this level of integration is found between O/Ses and their provided applications, the interface may or may not be available to third parties.
- o WPAD can be spoofed, allowing attackers to interpose a proxy and intercept traffic. This is a blocking issue for implementation.

3.3. Interception

The problems with manual configuration and proxy.pac/WPAD have led to the wide deployment of a third style of interposition; interception proxies.

Interception occurs when lower-layer protocols are configured to route HTTP traffic to a host other than the origin server for the URI in question. It requires no client configuration (hence its popularity over other methods). See [RFC3040] for an example of an interception-related protocol.

Interception is also strongly motivated when it is necessary to assure that the proxy is always used, e.g., to enforce policy.

Interception is problematic, however, because it is often done without the consent of either the end user or the origin server. This means that a response that appears to be coming from the origin server is actually coming from the intercepting proxy. This makes it difficult to support features like proxy authentication, as the

unexpected status code breaks many clients (e.g., non-interactive applications like software installers).

Furthermore, interception is a "layer violation"; i.e., misusing lower-layer protocols to enforce a higher-layer (often expressed as "layer 8") requirement.

In addition, as adoption of multi-path TCP (MPTCP) [RFC6824] increases, the ability of intercepting proxies to offer a consistent service degrades.

3.4. Configuration As Side Effect

More recently, it's become more common for a proxy to be interposed as a side effect of another choice by the user.

For example, the user might decide to add virus scanning - either as installed software, or a service that they configure from their provider - that is interposed as a proxy. Indeed, almost all desktop virus scanners and content filters operate in this fashion.

This approach has the merits of both being easy and obtaining explicit user consent. However, in some cases, the end user might not understand the consequences of use of the proxy, especially upon security and interoperability.

4. Second-Order Effects of Proxy Deployment

4.1. Proxies and HTTP

Deployment of proxies has an effect on the HTTP protocol itself. Because a proxy implements both a server and a client, any limitations or bugs in their implementation impact the protocol's use.

For example, HTTP has a defined mechanism for upgrading the protocol of a connection, to aid in the deployment of new versions of HTTP (such as HTTP/2) or completely different protocol (e.g., [RFC6455]).

However, operational experience has shown that a significant number of proxy implementations do not correctly implement it, leading to dangerous situations where two ends of a HTTP connection think different protocols are being spoken.

Another example is the Expect/100-continue mechanism in HTTP/1.1, which is often incorrectly implemented. Likewise, differences in support for trailers limits protocol extensions.

4.2. Proxies and TLS

It has become more common for Web sites to use TLS [RFC5246] in an attempt to avoid many of the problems above. Many have advocated use of TLS more broadly; for example, see the EFF's HTTPS Everywhere [https-everywhere] program, and SPDY's default use of TLS [I-D.mbelshe-httpbis-spdy].

However, doing so engenders a few problems.

Firstly, TLS as used on the Web is not a perfectly secure protocol, and using it to protect all traffic gives proxies a strong incentive to work around it, e.g., by deploying a certificate authority directly into browsers, or buying a sub-root certificate.

Secondly, it removes the opportunity for the proxy to inform the user agent of relevant information; for example, conditions of access, access denials, login interfaces, and so on. User Agents currently do not display any feedback from proxy, even in the CONNECT response (e.g., a 4xx or 5xx error), limiting their ability to have informed users of what's going on.

Finally, it removes the opportunity for services provided by a proxy that the end user might wish to opt into. For example, consider when a remote village shares a proxy server to cache content, thereby helping to overcome the limitations of their Internet connection. TLS-protected HTTP traffic cannot be cached by intermediaries, removing much of the benefit of the Web to what is arguably one of its most important target audiences.

It is now becoming more common for a proxy to man-in-the-middle TLS connections (see [tls-mitm] for an overview), to gain access to the application message flows. This represents a serious degradation in the trust infrastructure of the Web.

Worse is the situation where proxies provide a certificate where they inure the user to a certificate warning that they then need to ignore in order to receive service.

5. Principles for Consideration

Every HTTP connection has at least three major stakeholders; the user (through their agent), the origin server (possibly using gateways such as a CDN) and the networks between them.

Currently, the capabilities of these stakeholders are defined by how the Web is deployed. Most notably, networks sometimes change content. If they change it too much, origin servers will start using

encryption. Changing the way that HTTP operates therefore has the potential to re-balance the capabilities of the various stakeholders.

This section proposes several straw-man principles for consideration as the basis of those changes. Their sole purpose here is to provoke discussion.

5.1. Proxies Have a Legitimate Place

As illustrated above, there are many legitimate uses for proxies, and they are a necessary part of the architecture of the Web. While all uses of proxies are not legitimate - especially when they're interposed without the knowledge or consent of the end user and the origin - undesirable intermediaries (i.e., those that break the reasonable expectations of other stakeholders) are a small portion of those deployed used.

Note that while proxies have a legitimate place, it does not imply that they are an equal stakeholder to other parties in all ways; e.g., they do not have a natural right to access encrypted content, for example.

5.2. Security Should be Encouraged

Any solution needs to give all stakeholders - end users, networks and origin servers - a strong incentive towards security.

This has subtle implications. If networks are disempowered disproportionately, they might react by blocking secure connections, discouraging origin servers (who often have even stronger profit incentives) from deploying encryption, which would result in a net loss of security.

On the other hand, if networks are given carte blanche, it can destroy trust in the Web altogether. In particular, making it too easy to interpose a proxy (even if the user is "informed" by clicking through a dialogue) degrades the infrastructure in an unacceptable way.

5.3. Users Need to be Informed of Proxies

When a proxy is interposed, the user needs to be informed about it, so they have the opportunity to change their configuration (e.g., attempt to introduce encryption), or not use the network at all.

Proxies also need to be strongly authenticated; i.e., users need to be able to verify who the proxy is.

5.4. Users Need to be able to Tunnel through Proxies

When a proxy is interposed, the user needs to be able to tunnel any request through it without its content (or that of the response) being exposed to the proxy.

This includes both "https://" and "http://" URIs.

5.5. Proxies Can say "No"

A proxy can refuse to forward any request. Currently, the granularity of that "no" is per-URI for unencrypted requests, and per-IP (perhaps per-SNI) for encrypted requests.

5.6. Changes Need to be Detectable

Any changes to the message body, request URI, method, status code, or representation header fields of an HTTP message need to be detectable by the origin server or user agent, as appropriate, if they desire it.

This allows a proxy to be trusted, but its integrity to be verified.

5.7. Proxies Need to be Easy

It must be possible to configure a proxy extremely easily; the adoption of interception over proxy.pac/WPAD illustrates this very clearly.

5.8. Proxies Need to Communicate to Users

There are many situations where a proxy needs to communicate with the end user; for example, to gather network authentication credentials, communicate network policy, report that access to content has been denied, and so on.

Currently, HTTP has poor facilities for doing so. The proxy authentication mechanism is extremely limited, and while there are a few status codes that are defined as being from a proxy rather than the origin, they do not cover all necessary situations.

The Warning header field ([RFC7234], Section 5.5) was designed as a very limited form of communication between proxies and end users, but it has not been widely adopted, nor exposed by User Agents.

Importantly, proxies also need a limited communication channel when TLS is in use, for similar purposes.

Equally as important, the communication needs to clearly come from the proxy, rather than the origin, and be strongly authenticated.

5.9. Users Require Simple Interfaces

While some users are sophisticated in their understanding of Web security, they are in a vanishingly small minority. The concepts and implications of many decisions regarding security are subtle, and require an understanding of how the Web works; describing these trade-offs in a modal dialogue box that gets in the way of the content the user wants has been proven not to work.

Similarly, while some Web publishers are sophisticated regarding security, the vast majority are not (as can be proven by the prevalence of cross-site scripting attacks).

Therefore, any changes cannot rely upon perfect understanding by these parties, or even any great effort upon their part. This implies that user interface will be one of the biggest challenges faced, both in the browser and for any changes server-side.

Notably, the most widely understood indicator of security today is the "lock icon" that shows when a connection is protected by TLS. Any erosion of the commonly-understood semantics of that indicator, as well as "https://" URIs, is likely to be extremely controversial, because it changes the already-understood security properties of the Web.

Another useful emerging convention is that of "Incognito" or "private" mode, where the end user has requested enhanced privacy and security. This might be used to introduce higher requirements for the interposition of intermediaries, or even to prohibit their use without full encryption.

5.10. User Agents Are Diverse

HTTP is used in a wide variety of environments. As such there can be no assumption that a user is sitting on the other end to interpret information or answer questions from proxies.

5.11. RFC2119 Doesn't Define Reality

It's very tempting for a committee to proclaim that proxies "MUST" do this and "SHOULD NOT" do that, but the reality is that the proxies, like any other actor in a networked system, will do what they can, not what they're told to do, if they have an incentive to do it.

Therefore, it's not enough to say that (for example), "proxies have to honor no-transform" as HTTP/1.1 does. Instead, the protocol needs to be designed in a way so that either transformations aren't possible, or if they are, they can be detected (with appropriate handling by User Agents defined).

5.12. It Needs to be Deployable

Any improvements to the proxy ecosystem **MUST** be incrementally deployable, so that existing clients can continue to function.

6. Potential Areas to Investigate

Finally, this section lists some areas of potential future investigation, bearing the principles suggested above in mind.

6.1. Improving Proxy.Pac

Many of the flaws in proxy.pac can be fixed by careful specification and standardization, with active participation by both implementers and those that deploy it.

6.2. TLS Errors for Proxies

HTTP's use of TLS [RFC2818] currently offers no way for an interception proxy to communicate with the user agent on its own behalf. This might be necessary for network authentication, notification of filtering by hostname, etc.

The challenge in defining such a mechanism is avoiding the opening of new attack vectors; if unauthenticated content can be served as if it were from the origin server, or the user can be encouraged to "click through" a dialog, it has severe security implications. As such, the user experience would need to be carefully considered.

6.3. HTTP Errors for Proxies

HTTP currently defines two status codes that are explicitly generated by a proxy:

- o 504 Gateway Timeout ([RFC7231], Section 6.6.5) - when a proxy (or gateway) times out going forward
- o 511 Network Authentication Required ([RFC6585], Section 6) - when authentication information is necessary to access the network

It might be interesting to discuss whether a separate user experience can be formed around proxy-specific status codes, along with the definition of new ones as necessary.

6.4. TLS for Proxy Connections

While TLS can be used end-to-end for "https://" URIs, support for connecting to a proxy itself using TLS (e.g., for "http://" URIs) is spotty. Using a proxy without strong proof of its identity introduces security issues, and if a proxy can legitimately insert itself into communication, its identity needs to be verifiable.

6.5. Improved Network Information

Many of the use cases for proxies that modify content is transcoding or otherwise adapting that which is too "heavy" for the network it is transiting through.

If network operators made better, more fine-grained and timely information about their operational characteristics freely available, endpoints (server and client) could adapt requests and responses to reflect it, thereby removing the need for intermediation.

6.6. Improving Trust

Currently, it is possible to exploit the mismatched incentives and other flaws in the CA system to cause a browser to trust a proxy as authoritative for a "https://" URI without full user knowledge. This needs to be remedied; otherwise, proxies will continue to man-in-the-middle TLS.

6.7. HTTP Signatures

Signatures for HTTP content - both requests and responses - have been discussed on and off for some time.

Of particular interest here, signed responses would allow a user-agent to verify that the origin's content has not been modified in transit, whilst still allowing it to be cached by intermediaries.

Likewise, if header values can be signed, the caching policy (as expressed by Cache-Control, Date, Last-Modified, Age, etc.) can be signed, meaning it can be verified as being adhered to.

Note that properly designed, a signature mechanism could work over TLS, separating the trust relationship between the UA and the origin server and that of the UA and its proxy (with appropriate consent).

There are significant challenges in designing a robust, widely-deployable HTTP signature mechanism. One of the largest is an issue of user interface - what ought the UA do when encountering a bad signature?

7. Security Considerations

Plenty of them, I suspect.

8. Acknowledgements

This document benefits from conversations and feedback from many people, including Amos Jeffries, Willy Tarreau, Patrick McManus, Roberto Peon, Guy Podjarny, Eliot Lear, Brad Hill, Martin Nilsson and Julian Reschke.

9. References

9.1. Normative References

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

9.2. Informative References

- [I-D.mbelshe-httpbis-spdy]
Belshe, M. and R. Peon, "SPDY Protocol", draft-mbelshe-httpbis-spdy-00 (work in progress), February 2012.
- [RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.
- [RFC3040] Cooper, I., Melve, I., and G. Tomlinson, "Internet Web Replication and Caching Taxonomy", RFC 3040, January 2001.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.
- [RFC6455] Fette, I. and A. Melnikov, "The WebSocket Protocol", RFC 6455, December 2011.
- [RFC6585] Nottingham, M. and R. Fielding, "Additional HTTP Status Codes", RFC 6585, April 2012.
- [RFC6824] Ford, A., Raiciu, C., Handley, M., and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses", RFC 6824, January 2013.

- [RFC7230] Fielding, R. and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, June 2014.
- [RFC7231] Fielding, R. and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, June 2014.
- [RFC7234] Fielding, R., Nottingham, M., and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Caching", RFC 7234, June 2014.
- [W3C.NOTE-ct-guidelines-20101026]
Rabin, J., "Guidelines for Web Content Transformation Proxies 1.0", World Wide Web Consortium NOTE NOTE-ct-guidelines-20101026, October 2010, <<http://www.w3.org/TR/2010/NOTE-ct-guidelines-20101026>>.
- [https-everywhere]
EFF, ., "HTTPS Everywhere", 2013, <<https://www.eff.org/https-everywhere>>.
- [proxypac]
various, ., "Proxy Auto-Config", 2013, <http://en.wikipedia.org/wiki/Proxy_auto-config>.
- [tls-mitm]
Jarmoc, J., "SSL/TLS Interception Proxies and Transitive Trust", 2012, <https://www.grc.com/miscfiles/HTTPS_Interception_Proxies.pdf>.
- [wpad] Cohen, J., "Web Proxy Auto-Discovery Protocol", 1999, <<http://tools.ietf.org/html/draft-ietf-wrec-wpad-01>>.

Author's Address

Mark Nottingham

Email: mnot@mnot.netURI: <http://www.mnot.net/>

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: November 21, 2014

M. Nottingham

M. Thomson
Mozilla
May 20, 2014

Opportunistic Encryption for HTTP URIs
draft-nottingham-http2-encryption-03

Abstract

This describes how "http" URIs can be accessed using Transport Layer Security (TLS) to mitigate pervasive monitoring attacks.

Status of This Memo

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

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

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

This Internet-Draft will expire on November 21, 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 |
| 1.1. Goals and Non-Goals | 2 |
| 1.2. Notational Conventions | 3 |
| 2. Using HTTP URIs over TLS | 3 |
| 3. Server Authentication | 3 |
| 4. Interaction with "https" URIs | 4 |
| 5. Requiring Use of TLS | 4 |
| 5.1. The HTTP-TLS Header Field | 5 |
| 5.2. Operational Considerations | 6 |
| 6. Security Considerations | 6 |
| 6.1. Security Indicators | 7 |
| 6.2. Downgrade Attacks | 7 |
| 6.3. Privacy Considerations | 7 |
| 7. References | 7 |
| 7.1. Normative References | 7 |
| 7.2. Informative References | 8 |
| Appendix A. Acknowledgements | 8 |
| Authors' Addresses | 8 |

1. Introduction

This document describes a use of HTTP Alternative Services [I-D.ietf-httpbis-alt-svc] to decouple the URI scheme from the use and configuration of underlying encryption, allowing a "http" URI to be accessed using TLS [RFC5246] opportunistically.

Currently, "https" URIs requires acquiring and configuring a valid certificate, which means that some deployments find supporting TLS difficult. Therefore, this document describes a usage model whereby sites can serve "http" URIs over TLS without being required to support strong server authentication.

A mechanism for limiting the potential for active attacks is described in Section 5. This provides clients with additional protection against them for a period after successfully connecting to a server using TLS. This does not offer the same level of protection as afforded to "https" URIs, but increases the likelihood that an active attack be detected.

1.1. Goals and Non-Goals

The immediate goal is to make the use of HTTP more robust in the face of pervasive passive monitoring [RFC7258].

A secondary goal is to limit the potential for active attacks. It is not intended to offer the same level of protection as afforded to

"https" URIs, but instead to increase the likelihood that an active attack can be detected.

A final (but significant) goal is to provide for ease of implementation, deployment and operation. This mechanism should have a minimal impact upon performance, and should not require extensive administrative effort to configure.

1.2. 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].

2. Using HTTP URIs over TLS

An origin server that supports the resolution of HTTP URIs can indicate support for this specification by providing an alternative service advertisement [I-D.ietf-httpbis-alt-svc] for a protocol identifier that uses TLS, such as "h2" [I-D.ietf-httpbis-http2].

A client that receives such an advertisement MAY direct future requests for the associated origin to the identified service (as specified by [I-D.ietf-httpbis-alt-svc]).

A client that places the importance of passive protections over performance might choose to withhold requests until an encrypted connection is available. However, if such a connection cannot be successfully established, the client MAY resume its use of the cleartext connection.

A client can also explicitly probe for an alternative service advertisement by sending a request that bears little or no sensitive information, such as one with the OPTIONS method. Clients with expired alternative services information could make a similar request in parallel to an attempt to contact an alternative service, to minimize the delays that might be incurred by failing to contact the alternative service.

3. Server Authentication

There are no existing expectations with respect to cryptographically strong server authentication when it comes to resolving HTTP URIs. Establishing it, as described in [RFC2818], creates a number of operational challenges. For these reasons, server authentication is not mandatory for HTTP URIs when using the mechanism described in this specification.

When connecting to an alternative service for an "http" URI, clients are required to perform the server authentication procedure described in Section 3.1 of [RFC2818]. The server certificate, if one is proffered by the alternative service, is not necessarily checked for validity, expiration, issuance by a trusted certificate authority or matched against the name in the URI. Therefore, the alternative service MAY provide any certificate, or even select TLS cipher suites that do not include authentication.

A client MAY perform additional checks on the certificate that it is offered (if the server does not select an unauthenticated TLS cipher suite). For instance, a client could examine the certificate to see if it has changed over time.

In order to retain the authority properties of "http" URIs, and as stipulated by [I-D.ietf-httpbis-alt-svc], clients MUST NOT use alternative services that identify a host other than that of the origin, unless the alternative service indication itself is strongly authenticated. This is not currently possible for "http" URIs on cleartext transports.

4. Interaction with "https" URIs

An alternative service that is discovered to support "http" URIs might concurrently support "https" URIs, because HTTP/2 permits the sending of requests for multiple origins (see [RFC6454]) on the one connection. Therefore, when using alternative services, both HTTP and HTTPS URIs might be sent on the same connection.

"https" URIs rely on server authentication. Therefore, if a connection is initially created without authenticating the server, requests for "https" resources cannot be sent over that connection until the server certificate is successfully authenticated. Section 3.1 of [RFC2818] describes the basic mechanism, though the authentication considerations in [I-D.ietf-httpbis-alt-svc] could also apply.

Connections that are established without any means of server authentication (for instance, the purely anonymous TLS cipher suites), cannot be used for "https" URIs.

5. Requiring Use of TLS

Editors' Note: this is a very rough take on an approach that would provide a limited form of protection against downgrade attack. It's unclear at this point whether the additional effort (and modest operational cost) is worthwhile.

The mechanism described in this specification is trivial to mount an active attack against, for two reasons:

- o A client that doesn't perform authentication an easy victim of server impersonation, through man-in-the-middle attacks.
- o A client that is willing to use cleartext to resolve the resource will do so if access to any TLS-enabled alternative services is blocked at the network layer.

Given that the primary goal of this specification is to prevent passive attacks, these are not critical failings (especially considering the alternative - HTTP over cleartext). However, a modest form of protection against active attacks can be provided for clients on subsequent connections.

When an alternate service is able to commit to providing service for a particular origin over TLS for a bounded period of time, clients can choose to rely upon its availability, failing when it cannot be contacted. Effectively, this makes the alternative service "sticky" in the client.

One drawback with this approach is that clients need to strongly authenticate the alternative service to act upon such a commitment; otherwise, an attacker could create a persistent denial of service.

5.1. The HTTP-TLS Header Field

A alternative service can make this commitment by sending a "HTTP-TLS" header field:

```
HTTP-TLS      = 1#parameter
```

When it appears in a HTTP response from a strongly authenticated alternative service, this header field indicates that the availability of the origin through TLS-protected alternative services is "sticky", and that the client MUST NOT fall back to cleartext protocols while this information is considered fresh.

For example:


```
HTTP/1.1 200 OK
Content-Type: text/html
Cache-Control: 600
Age: 30
Date: Thu, 1 May 2014 16:20:09 GMT
HTTP-TLS: ma=3600
```

Note that the commitment is not bound to a particular alternative service; clients SHOULD use other alternative services that they become aware of, as long as the requirements regarding authentication and avoidance of cleartext protocols are met.

When this header field appears in a response, clients MUST strongly authenticate the alternative service, as described in Section 3.1 of [RFC2818], noting the additional requirements in [I-D.ietf-httpbis-alt-svc]. The header field MUST be ignored if strong authentication fails.

Persisted information expires after a period determined by the value of the "ma" parameter. See Section 4.2.3 of [I-D.ietf-httpbis-p6-cache] for details of determining response age.

ma-parameter = delta-seconds

Requests for an origin that has a persisted, unexpired value for "HTTP-TLS" MUST fail if they cannot be made over an authenticated TLS connection.

5.2. Operational Considerations

To avoid situations where a persisted value of "HTTP-TLS" causes a client to be unable to contact a site, clients SHOULD limit the time that a value is persisted for a given origin. A hard limit might be set to a month. A lower limit might be appropriate for initial observations of "HTTP-TLS"; the certainty that a site has set a correct value - and the corresponding limit on persistence - can increase as the value is seen more over time.

Once a server has indicated that it will support authenticated TLS, a client MAY use key pinning [I-D.ietf-websec-key-pinning] or any other mechanism that would otherwise be restricted to use with HTTPS URIs, provided that the mechanism can be restricted to a single HTTP origin.

6. Security Considerations

6.1. Security Indicators

User Agents MUST NOT provide any special security indicia when an "http" resource is acquired using TLS. In particular, indicators that might suggest the same level of security as "https" MUST NOT be used (e.g., using a "lock device").

6.2. Downgrade Attacks

A downgrade attack against the negotiation for TLS is possible. With the "HTTP-TLS" header field, this is limited to occasions where clients have no prior information (see Section 6.3), or when persisted commitments have expired.

For example, because the "Alt-Svc" header field [I-D.ietf-httpbis-alt-svc] likely appears in an unauthenticated and unencrypted channel, it is subject to downgrade by network attackers. In its simplest form, an attacker that wants the connection to remain in the clear need only strip the "Alt-Svc" header field from responses.

As long as a client is willing to use cleartext TCP to contact a server, these attacks are possible. The "HTTP-TLS" header field provides an imperfect mechanism for establishing a commitment. The advantage is that this only works if a previous connection is established where an active attacker was not present. A continuously present active attacker can either prevent the client from ever using TLS, or offer a self-signed certificate. This would prevent the client from ever seeing the "HTTP-TLS" header field, or if the header field is seen, from successfully validating and persisting it.

6.3. Privacy Considerations

Clients that persist state for origins can be tracked over time based on their use of this information. Persisted information can be cleared to reduce the ability of servers to track clients. A browser client MUST clear persisted all alternative service information when clearing other origin-based state (i.e., cookies).

7. References

7.1. Normative References

[I-D.ietf-httpbis-alt-svc]
Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", draft-ietf-httpbis-alt-svc-01 (work in progress), April 2014.

- [I-D.ietf-httpbis-http2]
Belshe, M., Peon, R., and M. Thomson, "Hypertext Transfer Protocol version 2", draft-ietf-httpbis-http2-12 (work in progress), April 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-websec-key-pinning]
Evans, C., Palmer, C., and R. Sleevi, "Public Key Pinning Extension for HTTP", draft-ietf-websec-key-pinning-13 (work in progress), May 2014.
- [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.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

7.2. Informative References

- [RFC6454] Barth, A., "The Web Origin Concept", RFC 6454, December 2011.
- [RFC7258] Farrell, S. and H. Tschofenig, "Pervasive Monitoring Is an Attack", BCP 188, RFC 7258, May 2014.

Appendix A. Acknowledgements

Thanks to Patrick McManus, Eliot Lear, Stephen Farrell, Guy Podjarny, Stephen Ludin, Erik Nygren, Paul Hoffman, Adam Langley, Eric Rescorla and Richard Barnes for their feedback and suggestions.

Authors' Addresses

Mark Nottingham

Email: mnot@mnot.net

URI: <http://www.mnot.net/>

Martin Thomson
Mozilla

Email: martin.thomson@gmail.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: April 24, 2014

V. Narayanan
Google
October 21, 2013

Explicit Proxying in HTTP - Problem Statement And Goals
draft-vidya-httpbis-explicit-proxy-ps-00

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.

Abstract

This document describes the issues with HTTP proxies for TLS protected traffic and motivates the need for explicit proxying capability in HTTP. It also presents the goals that such a solution would need to satisfy and some example solution directions.

Status of This Memo

This Internet-Draft is submitted to IETF 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 April 24, 2014.

Copyright Notice

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

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document.

Table of Contents

| | |
|---|---|
| 1. Introduction | 2 |
| 2. Motivation | 3 |
| 3. Proxying Needs Today | 3 |
| 4. Proxy Configurations Today | 4 |
| 5. Problems With Proxies Today | 5 |
| 6. Explicit HTTP Proxying | 5 |
| 6.1. Assumptions | 6 |
| 6.1.1. On Users, Intermediaries And Content Providers | 6 |
| 6.1.2. On Today's Practices | 6 |
| 6.2. Goals | 7 |
| 7. Potential Solution Directions | 8 |
| 7.1. Do Nothing | 8 |
| 7.2. Signed Policy Per Origin | 8 |
| 7.3. Explicit Proxy Detection using HTTP/TLS | 8 |
| 7.4. Object Level Security in HTTP | 9 |
| 7.5. TLS Origin Cert Exchanges | 9 |
| 8. Security Considerations | 9 |
| 9. IANA Considerations | 9 |
| 10. Acknowledgments | 9 |
| 11. Normative References | 9 |
| Author's Address | 9 |

1. Introduction

Web proxies that are present in the communication path between clients and servers are fairly common practice. There are many reasons one may employ a proxy, but the commonly deployed scenarios today are at odds with client to server privacy. While in almost all cases, some kind of user consent is received to carry traffic through proxies, such consent is fairly vague and the user is often unaware about the extent to which proxies have visibility into their communications. In some cases, such visibility may be construed as an unacceptable violation of privacy.

This document describes the types of proxies and the issues with the currently used models. It makes a case for legitimizing the use of

proxies while providing sufficient transparency to the endpoints. Towards that end, it also sets goals towards designing such an explicit proxying mechanism for http communication. Note that the scope of this document is limited to HTTPS only - HTTP communication not protected by TLS is out of scope for this discussion.

2. Motivation

The move to securing http connections is often to provide a confidential channel for exchange of information between a client and server. In the presence of a proxy, a secure channel may be perceived to be end-to-end when it is in fact not the case. To fix this, proxies must be visible to the communicating endpoints. However, without an interoperable solution, explicit proxying of connections becomes an issue. The motivations to make proxying explicit include:

- o Making secure communications possible for users
- o Allowing endpoints to choose not to communicate in the presence of an intermediary
- o Ensuring that a proxy is authorized to be in the path
- o Allowing detection of modified content
- o Allowing the ability to cache content in intermediate entities

3. Proxying Needs Today

There are number of reasons proxies have been deployed in networks. Some of these reasons include:

- o Policy Enforcement

Authentication and bandwidth policies may be often enforced using a proxy. This allows the model of having conditional connectivity or limits on connectivity such as may be observed in a hotel or an airport hotspot, among other places. For TLS protected connections, this type of policy enforcement becomes difficult (the connections just fail until the user finds a way to authenticate with the proxy). But, it may be useful to support this using explicit proxying techniques for a better experience.

- o Caching

Very widely used on the Internet, caching proxies allow serving of content from topologically closer sources in order to preserve network resources and improve user experience.

- o Content Modification

Certain networks employ content modifying proxies that generally modify content to improve network and overall user experience. For instance, proxies in mobile networks may need to modify content to change the codec for sake of older devices. However, sometimes, content modification proxies have also been known to make high definition content unavailable - while arguably this may be in the best interest in balancing the overall health of the network (a bad network isn't good for any user), this is also highly debatable.

- o Content Filtering

Content filtering proxies may prevent malware, but may also prevent access to sites that are in violation of the administrative policies. This filtering may be by size, type or subject matter.

- o Content Inspection

Some proxies may be installed with a need to inspect and flag inappropriate content (e.g., in schools).

As can be seen, some of the proxies need access to the content, while others do not.

4. Proxy Configurations Today

Proxies used to be configured manually by having the users specify the proxy information in the browser settings. However, due to the difficulty in effectively implementing this approach (specifically that the user needed to be involved when he/she moved out of the proxy's network), two proxying models have evolved. One is the group configuration policies that are used in enterprises, where a device that is part of an enterprise gets subjected to the enterprise proxying policies by pre configuration.

Another is the model of "interception" or "transparent" proxies became widely popular and is also in a fairly large use today. In the "transparent" proxy model, neither endpoint knows about the interception. Transparent proxies can be employed in the presence of TLS (HTTPS) as well, when certificate pinning is disabled. Most deployments end up disabling certificate pinning so that proxying can be accomplished. The client machines are often configured with root certs that will allow it to accept the proxy generated ephemeral certificate for the server. Future configurations of proxies continues to be a problem and explicit mechanisms of configuring

proxies may be necessary to motivate the move away from interception proxies.

There are also tunneling proxies, where HTTP CONNECT may be used to tunnel the client requests to the server.

5. Problems With Proxies Today

The use of proxies leads to a number of privacy issues. To summarize:

- o The user often has no knowledge that their data exchanges are passing through an interception proxy that potentially has visibility to the actual content exchanged.
- o The server has no knowledge of the presence of the proxy and hence, cannot refuse to serve sensitive content over a proxied connection.
- o The weakened security model, when certificate pinning is disabled at a general level, allows inspection of content potentially by entities other than legitimate proxies that the user may be willing to give access to. This is especially true in enterprises with a multitude of platforms, devices and browsers, where explicit configuration of proxy certificates is an administrative burden.
- o The client can no longer authenticate the real server. Hence, the client has no influence on certificate revocation checks and any visible warnings to the user are made infeasible.
- o Client authentication (and hence mutual authentication) is made infeasible. Any server relying upon mutual authentication to establish a TLS connection cannot work with transparent proxies.
- o The user has no way of detecting whether content has been modified en route.
- o The client is susceptible to downgrade attacks, as it cannot do any policy enforcement on versions or algorithms.

With privacy becoming more and more important, it is important for us to support solutions that allow awareness of a privacy breach to both users and the servers, when that happens. To this effect, it is important that proxies be explicitly supported and detected.

6. Explicit HTTP Proxying

The problems illustrated in this document call for explicit knowledge of on-path proxies to the user and the content provider. The key assumptions and goals driving this target are stated below.

6.1. Assumptions

6.1.1. On Users, Intermediaries And Content Providers

- o Users configure proxies and forget about the configuration.
- o Users have limited control over provider installed certificates.
 - * Often, the user's only choice is to not sign up for service at all.
- o Users may not wish to have some or any of their communications intercepted, even when they are on a network for which they previously configured a proxy.
- o Intermediaries have various legitimate reasons for wanting to inspect traffic:
 - * Cache content
 - * Implement network traffic policies (e.g. legal compliance, malware detection, etc)
- o Content providers may not wish to serve certain content in anything less than an end-to-end secure fashion.

6.1.2. On Today's Practices

- o Many networks seem to leave the traffic on port 443 untouched and unblocked today, likely as a result of both the importance of the data and the relative rarity of communications using TLS. It is unclear how this might change as TLS protected traffic increases, if we continued to not have a solution for explicit proxying.
- o Entities which need to inspect traffic on port 443 today are forced to either block port 443 or to deploy an intercepting proxy and install root certs on all devices which may use the network. In the latter case, the deployed proxy impersonates both the content-provider to the user-agent, and the user-agent to the content-provider. Though there is work to allow users to detect these situations [DANE], support is not widespread.
- o Many, if not most, mobile devices using cellular networks use proxies and several of them act as transforming proxies.

- o Users and sites have only one mechanism for specifying point-to-point security policy for HTTP [RFC2616], which is the scheme of the URI identifying any particular resource.

6.2. Goals

These are the goals of a solution aimed at making proxying explicit in HTTP.

- o In the presence of a proxy, users' communications SHOULD at least use a channel that is point-to-point encrypted.
- o Users MUST be able to opt-out of communicating sensitive information over a channel which is not end-to-end private.
- o Content-providers MAY serve certain content only in an end-to-end confidential fashion.
- o Interception proxies MUST be precluded from intercepting secure communications between the user and the content-provider.
- o User-agents and servers MUST know when a transforming proxy is interposed in the communications channel.
- o User-agents MUST be able to detect when content requested with an https scheme has been modified by any intermediate entity.
- o Entities other than the server or user-agent SHOULD still be able to provide caching services.
- o User agents MUST be able to verify that the content is in fact served by the content provider.
- o A set of signaling semantics should exist which allows the content-provider and the user to have the appropriate level of security or privacy signaled per resource.
- o Ideally, HTTP URI semantics SHOULD NOT change, but if it does, it must remain backwards-compatible.
- o Configuration and deployment of proxies should be as easy as currently used solutions.
- o Introduction of explicit proxying MUST NOT require a flag day upgrade - in other words, it should work with existing client and content provider implementations during the transition.

7. Potential Solution Directions

There are several potential directions this work can take, some of which are clearly undesirable and some that are more viable. This section is not intended for an exhaustive description of each solution - rather, it is aimed at serving as a starting point for discussions. Note that more than one of these directions may need to be adopted and brought together for a complete solution - so, each section here is not intended to be standalone and complete.

7.1. Do Nothing

This is a scenario that continues to support interception proxies in current modes. The fundamental premise of this document is based on the fact that this is bad.

7.2. Signed Policy Per Origin

RFC6454 specifies a method by which there can be a signed policy per origin. However, such coarse granularity of providing a policy for an entire domain is often not useful and hence, this is rarely used in practice.

7.3. Explicit Proxy Detection using HTTP/TLS

Means of:

- o Explicit signaling of presence of proxy from user agent to server.
- o Signaling to indicate user preference for end-to-end secure communication
- o Signaling to indicate content unavailability via proxies
- o Verification of proxy identity to detect untrusted proxies
- o Serving interstitial pages to manage portals that enforce bandwidth, connectivity times, etc.

The above can be accomplished in a variety of ways, including HTTP/TLS error codes, HTTP2.0 proxy signaling semantics and HTTP/TLS exchange of proxy identities.

7.4. Object Level Security in HTTP

The ability to detect modified content is needed. Specifically:

- o Object level integrity protection of content by content provider
- o Object level encryption by content provider (optionally)

7.5. TLS Origin Cert Exchanges

The ability to exchange the true certificate chain of the server in TLS exchanges so that clients can make better decisions about servers.

8. Security Considerations

TBD

9. IANA Considerations

TBD

10. Acknowledgments

List of names here - TBD

11. Normative References

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

Author's Address

Vidya Narayanan
Google
1600 Amphitheatre Pkwy
Mountain View, CA
USA

Email: vn@google.com