HTTP Working Group
INTERNET-DRAFT
<draft-ietf-http-v11-spec-02.html>

Expires September 23, 1996

R. Fielding, UC Irvine
H. Frystyk, MIT/LCS
T. Berners-Lee, MIT/LCS
J. Gettys, DEC
Jeffrey C. Mogul, DEC
April 23, 1996

Hypertext Transfer Protocol -- HTTP/1.1

Status of this Memo

This document is an Internet-Draft. Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or made obsolete 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".

To learn the current status of any Internet-Draft, please check the "`1id-abstracts.txt`" listing contained in the Internet-Drafts Shadow Directories on `ftp.is.co.za` (Africa), `nic.nordu.net` (Europe), `munnari.oz.au` (Pacific Rim), `ds.internic.net` (US East Coast), or `ftp.isi.edu` (US West Coast).

Distribution of this document is unlimited. Please send comments to the HTTP working group at `<http-wg@cuckoo.hpl.hp.com>`. Discussions of the working group are archived at `<URL:http://www.ics.uci.edu/pub/ietf/http/>`. General discussions about HTTP and the applications which use HTTP should take place on the `<www-talk@w3.org>` mailing list.

> NOTE: This specification is for discussion purposes only. It is not claimed to represent the consensus of the HTTP working group, and contains a number of proposals that either have not been discussed or are controversial. The working group is discussing significant changes in many areas, including - support for caching, persistent connections, range retrieval, content negotiation, MIME compatibility, authentication, timing of the PUT operation.

Abstract

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, object-oriented protocol which can be used for many tasks, such as name servers and distributed object management systems, through extension of its request methods (commands). A feature of HTTP is the typing and negotiation of data representation, allowing systems to be built independently of the data being transferred.

HTTP has been in use by the World-Wide Web global information initiative since 1990. This specification defines the protocol referred to as "HTTP/1.1".

Note to Readers of This Document

This document is still organized to minimize changes from the previous draft, to ease reviewers work in finding new material (and because the editor has not had time to reorganize it).. However, the current organization is now quite poor for new readers of this document. We recommend that new readers of this document not read it in the current order of presentation, but may want to skip ahead after reading sections 1-9 and read sections 11, 12 13 and 14 before reading section 10 which defines the header field definitions. Section 10 itself is now also not in alphabetical order, again, to avoid renumbering sections to be able to easily compare between drafts.

If you are reading the version of this document showing revision markup, note that we've tried to preserve significant changes from the previous version, though a few changes may have slipped through unmarked. We make no guarantees that all changes have revision marks, though we've tried to preserve them as an aid to those who wish to check a specific change has been reflected in this draft.

Note that some sections are still marked as SLUSHY and a few are marked FLUID; these are still undergoing drafting.

Note that text in bold in the text are as yet incompletely resolved issues. Opinions are solicited…

Table of Contents

1. Introduction

1.1 Purpose

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. HTTP has been in use by the World-Wide Web global information initiative since 1990. The first version of HTTP, referred to as HTTP/0.9, was a simple protocol for raw data transfer across the Internet. HTTP/1.0, as defined by RFC xxxx [6], improved the protocol by allowing messages to be in the format of MIME-like entities, containing metainformation about the data transferred and modifiers on the request/response semantics. However, HTTP/1.0 does not sufficiently take into consideration the effect of hierarchical proxies and caching, the desire for persistent connections and virtual hosts, and a number of other details that slipped through the cracks of existing implementations. In addition, the proliferation of incompletely-implemented applications calling themselves “HTTP/1.0” has necessitated a protocol version change in order for two communicating applications to determine each other's true capabilities.

This specification defines the protocol referred to as “HTTP/1.1”. This protocol is backwards-compatible with HTTP/1.0, but includes more stringent requirements in order to ensure reliable implementation of its features.

Practical information systems require more functionality than simple retrieval, including search, front-end update, and annotation. HTTP allows an open-ended set of methods that indicate the purpose of a request. It builds on the discipline of reference provided by the Uniform Resource Identifier (URI) [3], as a location (URL) [4] or name (URN) [20], for indicating the resource on which a method is to be applied. Messages are passed in a format similar to that used by Internet Mail [9] and the Multipurpose Internet Mail Extensions (MIME) [7].

HTTP is also used as a generic protocol for communication between user agents and proxies/gateways to other Internet protocols, such as SMTP [16], NNTP [13], FTP [18], Gopher [2], and WAIS [10], allowing basic hypermedia access to resources available from diverse applications and simplifying the implementation of user agents.

1.2 Requirements

This specification uses the same words as RFC 1123 [8] for defining the significance of each particular requirement. These words are:

MUST
: This word or the adjective “required” means that the item is an absolute requirement of the specification.

SHOULD
: This word or the adjective “recommended” means that there may exist valid reasons in particular circumstances to ignore this item, but the full implications should be understood and the case carefully weighed before choosing a different course.

MAY
: This word or the adjective “optional” means that this item is truly optional. One vendor may choose to include the item because a particular marketplace requires it or because it enhances the product, for example; another vendor may omit the same item.

An implementation is not compliant if it fails to satisfy one or more of the MUST requirements for the protocols it implements. An implementation that satisfies all the MUST and all the SHOULD requirements for its protocols is said to be “unconditionally compliant”; one that satisfies all the MUST requirements but not all the SHOULD requirements for its protocols is said to be “conditionally compliant”.

1.3 Terminology

This specification uses a number of terms to refer to the roles played by participants in, and objects of, the HTTP communication.

connection
: A transport layer virtual circuit established between two application programs for the purpose of communication.

message
: The basic unit of HTTP communication, consisting of a structured sequence of octets matching the syntax defined in Section 4 and transmitted via the connection.

request
: An HTTP request message (as defined in Section 5).

response
: An HTTP response message (as defined in Section 6).

resource
: A network data object or service that can be identified by a URI (Section 3.2).

entity
: A particular representation, rendition, encoding, or presentation of a resource. Resources not supporting content negotiation are bound to a single entity. Resources supporting content negotiation are bound to a set of one or more entities, whose membership may vary over time.

entity instance
: The definite value of an entity at a given point in time. The HTTP protocol transfers entity instances in request or response messages. An entity instance is transferred as metainformation in the form of entity headers and content in the form of an entity body.

client
: An application program that establishes connections for the purpose of sending requests.

user agent
: The client which initiates a request. These are often browsers, editors, spiders (web-traversing robots), or other end user tools.

server
: An application program that accepts connections in order to service requests by sending back responses.

origin server
: The server on which a given resource resides or is to be created.

proxy
: An intermediary program which acts as both a server and a client for the purpose of making requests on behalf of other clients. Requests are serviced internally or by passing them, with possible translation, on to other servers. A proxy MUST interpret and, if necessary, rewrite a request message before forwarding it. Proxies are often used as client-side portals through network firewalls and as helper applications for handling requests via protocols not implemented by the user agent.

gateway
: A server which acts as an intermediary for some other server. Unlike a proxy, a gateway receives requests as if it were the origin server for the requested resource; the requesting client may not be aware that it is communicating with a gateway. Gateways are often used as server-side portals through network firewalls and as protocol translators for access to resources stored on non-HTTP systems.

tunnel
: A tunnel is an intermediary program which is acting as a blind relay between two connections. Once active, a tunnel is not considered a party to the HTTP communication, though the tunnel may have been initiated by an HTTP request. The tunnel ceases to exist when both ends of the relayed connections are closed. Tunnels are used when a portal is necessary and the intermediary cannot, or should not, interpret the relayed communication.

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

Any given program MAY be capable of being both a client and a server; our use of these terms refers only to the role being performed by the program for a particular connection, rather than to the program's capabilities in general. Likewise, any server MAY act as an origin server, proxy, gateway, or tunnel, switching behavior based on the nature of each request.

1.4 Overall Operation

The HTTP protocol is based on a request/response paradigm. A client sends a request to the server in the form of a request method, URI, and protocol version, followed by a MIME-like message containing request modifiers, client information, and possible body content over a connection with a server. The server responds with a status line, including the message's protocol version and a success or error code, followed by a MIME-like message containing server information, entity metainformation, and possible body content.

Most HTTP communication is initiated by a user agent and consists of a request to be applied to a resource on some origin server. In the simplest case, this may be accomplished via a single connection (v) between the user agent (UA) and the origin server (O).

```
   request chain ------------------------>
UA -------------------v------------------- O
   <----------------------- response chain
```

A more complicated situation occurs when one or more intermediaries are present in the request/response chain. There are three common forms of intermediary: proxy, gateway, and tunnel. A proxy is a forwarding agent, receiving requests for a URI in its absolute form, rewriting all or parts of the message, and forwarding the reformatted request toward the server identified by the URI. A gateway is a receiving agent, acting as a layer above some other server(s) and, if necessary, translating the requests to the underlying server's protocol. A tunnel acts as a relay point between two connections without changing the messages; tunnels are used when the communication needs to pass through an intermediary (such as a firewall) even when the intermediary cannot understand the contents of the messages.

```
   request chain -------------------------------------->
UA -----v----- A -----v----- B -----v----- C -----v----- O
   <------------------------------------- response chain
```

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 MUST pass through four separate connections. This distinction is important because some HTTP communication options may 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 may be engaged in multiple, simultaneous communications. For example, B may 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.

Any party to the communication which is not acting as a tunnel may employ an internal cache for handling requests. 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 which has not been cached by UA or A.

```
   request chain ---------->
UA -----v----- A -----v----- B - - - - - - C - - - - - - O
   <--------- response chain
```

Not all responses are cachable, and some requests may contain modifiers which place special requirements on cache behavior. HTTP requirements for cache behavior and cachable responses are defined in Section 13.

On the Internet, HTTP communication generally takes place over TCP/IP connections. The default port is TCP 80 [19], but other ports can be used. This does not preclude HTTP from being implemented on top of any other protocol on the Internet, or on other networks. HTTP only presumes a reliable transport; any protocol that provides such guarantees can be used; the mapping of the HTTP/1.1 request and response structures onto the transport data units of the protocol in question is outside the scope of this specification.

However, HTTP/1.1 implementations SHOULD implement persistent connections (See section 14). Both clients and servers MUST be capable of handling cases where either party closes the connection prematurely, due to user action, automated time-out, or program failure. In any case, the closing of the connection by either or both parties always terminates the current request, regardless of its status.

1.4 HTTP and MIME

HTTP/1.1 uses many of the constructs defined for MIME, as defined in RFC 1521 [7]. Appendix C describes the ways in which the context of HTTP allows for different use of Internet Media Types than is typically found in Internet mail, and gives the rationale for those differences.

2. Notational Conventions and Generic Grammar

2.1 Augmented BNF

All of the mechanisms specified in this document are described in both prose and an augmented Backus-Naur Form (BNF) similar to that used by RFC 822 [9]. Implementers will need to be familiar with the notation in order to understand this specification. The augmented BNF includes the following constructs:

name = definition

The name of a rule is simply the name itself (without any enclosing "<" and ">") and is separated from its definition by the equal character "=". Whitespace is only significant in that indentation of continuation lines is used to indicate a rule definition that spans more than one line. Certain basic rules are in uppercase, such as SP, LWS, HT, CRLF, DIGIT, ALPHA, etc. Angle brackets are used within definitions whenever their presence will facilitate discerning the use of rule names.

"literal"

Quotation marks surround literal text. Unless stated otherwise, the text is case-insensitive.

rule1 | rule2

Elements separated by a bar ("|") are alternatives, e.g., "yes | no" will accept yes or no.

(rule1 rule2)

Elements enclosed in parentheses are treated as a single element. Thus, "(elem (foo | bar) elem)" allows the token sequences "elem foo elem" and "elem bar elem".

*rule

The character "*" preceding an element indicates repetition. The full form is "<n>*<m>element" indicating at least <n> and at most <m> occurrences of element. Default values are 0 and infinity so that "*(element)" allows any number, including zero; "1*element" requires at least one; and "1*2element" allows one or two.

[rule]

Square brackets enclose optional elements; "[foo bar]" is equivalent to "*1(foo bar)".

N rule

Specific repetition: "<n>(element)" is equivalent to "<n>*<n>(element)"; that is, exactly <n> occurrences of (element). Thus 2DIGIT is a 2-digit number, and 3ALPHA is a string of three alphabetic characters.

#rule

A construct "#" is defined, similar to "*", for defining lists of elements. The full form is

"<n>#<m>element" indicating at least <n> and at most <m> elements, each separated by one or more commas (",") and optional linear whitespace (LWS). This makes the usual form of lists very easy; a rule such as "(*LWS element *(*LWS "," *LWS element))" can be shown as "1#element". Wherever this construct is used, null elements are allowed, but do not contribute to the count of elements present. That is, "(element), , (element)" is permitted, but counts as only two elements. Therefore, where at least one element is required, at least one non-null element MUST be present. Default values are 0 and infinity so that "#(element)" allows any number, including zero; "1#element" requires at least one; and "1#2element" allows one or two.

`; comment`

A semi-colon, set off some distance to the right of rule text, starts a comment that continues to the end of line. This is a simple way of including useful notes in parallel with the specifications.

`implied *LWS`

The grammar described by this specification is word-based. Except where noted otherwise, linear whitespace (LWS) can be included between any two adjacent words (token or quoted-string), and between adjacent tokens and delimiters (tspecials), without changing the interpretation of a field. At least one delimiter (tspecials) MUST exist between any two tokens, since they would otherwise be interpreted as a single token. However, applications SHOULD attempt to follow "common form" when generating HTTP constructs, since there exist some implementations that fail to accept anything beyond the common forms.

2.2 Basic Rules

The following rules are used throughout this specification to describe basic parsing constructs. The US-ASCII coded character set is defined by [21].

```
OCTET          = <any 8-bit sequence of data>
CHAR           = <any US-ASCII character (octets 0 - 127)>
UPALPHA        = <any US-ASCII uppercase letter "A".."Z">
LOALPHA        = <any US-ASCII lowercase letter "a".."z">
ALPHA          = UPALPHA | LOALPHA
DIGIT          = <any US-ASCII digit "0".."9">
CTL            = <any US-ASCII control character
                 (octets 0 - 31) and DEL (127)>
CR             = <US-ASCII CR, carriage return (13)>
LF             = <US-ASCII LF, linefeed (10)>
SP             = <US-ASCII SP, space (32)>
HT             = <US-ASCII HT, horizontal-tab (9)>
<">            = <US-ASCII double-quote mark (34)>
```

HTTP/1.1 defines the octet sequence CR LF as the end-of-line marker for all protocol elements except the Entity-Body (see Appendix B for tolerant applications). The end-of-line marker within an Entity-Body is defined by its associated media type, as described in Section 3.7.

```
CRLF           = CR LF
```

HTTP/1.1 headers can be folded onto multiple lines if the continuation line begins with a space or horizontal tab. All linear whitespace, including folding, has the same semantics as SP.

```
LWS            = [CRLF] 1*( SP | HT )
```

The `TEXT` rule is only used for descriptive field contents and values that are not intended to be interpreted by the message parser. Words of `*TEXT` MAY contain octets from character sets other than US-ASCII only when encoded according to the rules of RFC 1522 [14].

```
TEXT           = <any OCTET except CTLs,
                 but including LWS>
```

Recipients of header field `TEXT` containing octets outside the US-ASCII character set range MAY assume that they represent ISO-8859-1 characters if there is no other encoding indicated by an RFC 1522 mechanism.

Hexadecimal numeric characters are used in several protocol elements.

```
HEX            = "A" | "B" | "C" | "D" | "E" | "F"
               | "a" | "b" | "c" | "d" | "e" | "f" | DIGIT
```

Many HTTP/1.1 header field values consist of words separated by `LWS` or special characters. These special characters MUST be in a quoted string to be used within a parameter value.

```
word           = token | quoted-string

token          = 1*<any CHAR except CTLs or tspecials>

tspecials      = "(" | ")" | "<" | ">" | "@"
               | "," | ";" | ":" | "\" | <">
               | "/" | "[" | "]" | "?" | "="
               | "{" | "}" | SP | HT
```

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. In all other fields, parentheses are considered part of the field value.

```
comment        = "(" *( ctext | comment ) ")"
ctext          = <any TEXT excluding "(" and ")">
```

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

```
quoted-string  = ( <"> *(qdtext) <"> )

qdtext         = <any CHAR except <"> and CTLs,
                 but including LWS>
```

The backslash character (“\”) may be used as a single-character quoting mechanism only within quoted-string and comment constructs.

```
quoted-pair    = "\" CHAR
```

3. Protocol Parameters

3.1 HTTP Version

HTTP uses a “<major>.<minor>” numbering scheme to indicate versions of the protocol. The protocol versioning policy is intended to allow the sender to indicate the format of a message and its capacity for understanding further HTTP communication, rather than the features obtained via that communication. No change is made to the version number for the addition of message components which do not affect communication behavior or which only add to extensible field values. The <minor> number is incremented when the changes made to the protocol add features which do not change the general message parsing algorithm, but which may add to the message semantics and imply additional capabilities of the sender. The <major> number is incremented when the format of a message within the protocol is changed.

The version of an HTTP message is indicated by an `HTTP-Version` field in the first line of the message. If the protocol version is not specified, the recipient MUST assume that the message is in the simple HTTP/0.9 format [6].

```
HTTP-Version   = "HTTP" "/" 1*DIGIT "." 1*DIGIT
```

Note that the major and minor numbers SHOULD be treated as separate integers and that each MAY be incremented higher than a single digit. Thus, HTTP/2.4 is a lower version than HTTP/2.13, which in turn is lower than HTTP/12.3. Leading zeros SHOULD be ignored by recipients and never generated by senders.

Applications sending `Full-Request` or `Full-Response` messages, as defined by this specification, MUST include an `HTTP-Version` of “`HTTP/1.1`”. Use of this version number indicates that the sending application is at least conditionally compliant with this specification.

Proxy and gateway applications MUST be careful in forwarding requests that are received in a format different than that of the application's native HTTP version. Since the protocol version indicates the protocol capability of the sender, a proxy/gateway MUST never send a message with a version indicator which is greater than its native version; if a higher version request is received, the proxy/gateway MUST either downgrade the request version, respond with an error, or switch to tunnel behavior. Requests with a version lower than that of the application's native format MAY be upgraded before being forwarded; the proxy/gateway's response to that request MUST follow the server requirements listed above.

> Note: Converting between versions of HTTP may involve addition or deletion of headers required or forbidden by the version involved. It is likely more involved than just changing the version indicator.

3.2 Uniform Resource Identifiers

URIs have been known by many names: WWW addresses, Universal Document Identifiers, Universal Resource Identifiers [3], and finally the combination of Uniform Resource Locators (URL) [4] and Names (URN) [20]. As far as HTTP is concerned, Uniform Resource Identifiers are simply formatted strings which identify--via name, location, or any other characteristic--a network resource.

3.2.1 General Syntax

URIs in HTTP can be represented in absolute form or relative to some known base URI [11], depending upon the context of their use. The two forms are differentiated by the fact that absolute URIs always begin with a scheme name followed by a colon.

```
URI            = ( absoluteURI | relativeURI ) [ "#" fragment ]

absoluteURI    = scheme ":" *( uchar | reserved )

relativeURI    = net_path | abs_path | rel_path

net_path       = "//" net_loc [ abs_path ]
abs_path       = "/" rel_path
rel_path       = [ path ] [ ";" params ] [ "?" query ]

path           = fsegment *( "/" segment )
fsegment       = 1*pchar
segment        = *pchar

params         = param *( ";" param )
param          = *( pchar | "/" )

scheme         = 1*( ALPHA | DIGIT | "+" | "-" | "." )
net_loc        = *( pchar | ";" | "?" )
query          = *( uchar | reserved )
fragment       = *( uchar | reserved )

pchar          = uchar | ":" | "@" | "&" | "=" | "+"
uchar          = unreserved | escape
unreserved     = ALPHA | DIGIT | safe | extra | national

escape         = "%" HEX HEX
reserved       = ";" | "/" | "?" | ":" | "@" | "&" | "="
extra          = "!" | "*" | "'" | "(" | ")" | ","
safe           = "$" | "-" | "_" | "." | "+"
unsafe         = CTL | SP | <"> | "#" | "%" | "<" | ">"
national       = <any OCTET excluding ALPHA, DIGIT,
                 reserved, extra, safe, and unsafe>
```

For definitive information on URL syntax and semantics, see RFC 1738 [4] and RFC 1808 [11]. The BNF above includes `national` characters not allowed in valid URLs as specified by RFC 1738, since HTTP servers are not restricted in the set of `unreserved` characters allowed to represent the `rel_path` part of addresses, and HTTP proxies may receive requests for URIs not defined by RFC 1738.

The HTTP protocol does not place any a-priori limit on the length of a URI. Servers MUST be able to handle the URI of any resource they serve, and SHOULD be able to handle URIs of unbounded length if they provide GET-based forms that could generate such URIs. A server SHOULD return a status code of

```
414 Request-URI Too Large
```

if a URI is longer than the server can handle. See section 9.4.

> Note: Servers SHOULD be cautious about depending on URI lengths above 255 bytes, because some older client or proxy implementations may not properly support these.

All client and proxy implementations MUST be able to handle a URI of any finite length.

3.2.2 http URL

The “http” scheme is used to locate network resources via the HTTP protocol. This section defines the scheme-specific syntax and semantics for http URLs.

```
http_URL       = "http:" "//" host [ ":" port ] [ abs_path ]

host           = <A legal Internet host domain name
                  or IP address (in dotted-decimal form),
                  as defined by Section 2.1 of RFC 1123>

port           = *DIGIT
```

If the `port` is empty or not given, port 80 is assumed. The semantics are that the identified resource is located at the server listening for TCP connections on that `port` of that `host`, and the `Request-URI` for the resource is `abs_path`. The use of IP addresses in URL's SHOULD be avoided whenever possible. See RFC 1900[24]. If the `abs_path` is not present in the URL, it MUST be given as “/” when used as a `Request-URI` for a resource (Section 5.1.2).

> Note: Although the HTTP protocol is independent of the transport layer protocol, the http URL only identifies resources by their TCP location, and thus non-TCP resources MUST be identified by some other URI scheme.

The canonical form for “http” URLs is obtained by converting any `UPALPHA` characters in `host` to their `LOALPHA` equivalent (hostnames are case-insensitive), eliding the `[ ":" port ]` if the port is 80, and replacing an empty `abs_path` with “/”.

3.3 Date/Time Formats

3.3.1 Full Date

HTTP applications have historically allowed three different formats for the representation of date/time stamps:

```
Sun, 06 Nov 1994 08:49:37 GMT    ; RFC 822, updated by RFC 1123
Sunday, 06-Nov-94 08:49:37 GMT   ; RFC 850, made obsolete by RFC 1036
Sun Nov  6 08:49:37 1994         ; ANSI C's asctime() format
```

The first format is preferred as an Internet standard and represents a fixed-length subset of that defined by RFC 1123 [8] (an update to RFC 822 [9]). The second format is in common use, but is based on the obsolete RFC 850 [12] date format and lacks a four-digit year. HTTP/1.1 clients and servers that parse the date value MUST accept all three formats, though they MUST only generate the RFC 1123 format for representing date/time stamps in HTTP message fields.

> Note: Recipients of date values are encouraged to be robust in accepting date values that may have been generated by non-HTTP applications, as is sometimes the case when retrieving or posting messages via proxies/gateways to SMTP or NNTP.

All HTTP date/time stamps MUST be represented in Universal Time (UT), also known as Greenwich Mean Time (GMT), without exception. This is indicated in the first two formats by the inclusion of “GMT” as the three-letter abbreviation for time zone, and SHOULD be assumed when reading the asctime format.

```
HTTP-date      = rfc1123-date | rfc850-date | asctime-date

rfc1123-date   = wkday "," SP date1 SP time SP "GMT"
rfc850-date    = weekday "," SP date2 SP time SP "GMT"
asctime-date   = wkday SP date3 SP time SP 4DIGIT

date1          = 2DIGIT SP month SP 4DIGIT
                 ; day month year (e.g., 02 Jun 1982)
date2          = 2DIGIT "-" month "-" 2DIGIT
                 ; day-month-year (e.g., 02-Jun-82)
date3          = month SP ( 2DIGIT | ( SP 1DIGIT ))
                 ; month day (e.g., Jun  2)

time           = 2DIGIT ":" 2DIGIT ":" 2DIGIT
                 ; 00:00:00 - 23:59:59

wkday          = "Mon" | "Tue" | "Wed"
               | "Thu" | "Fri" | "Sat" | "Sun"

weekday        = "Monday" | "Tuesday" | "Wednesday"
               | "Thursday" | "Friday" | "Saturday" | "Sunday"

month          = "Jan" | "Feb" | "Mar" | "Apr"
               | "May" | "Jun" | "Jul" | "Aug"
               | "Sep" | "Oct" | "Nov" | "Dec"
```

Note: HTTP requirements for the date/time stamp format apply only to their usage within the protocol stream. Clients and servers are not required to use these formats for user presentation, request logging, etc.

Additional rules for requirements on parsing and representation of dates and other potential problems with date representations include:

- HTTP/1.1 clients and caches should assume that an RFC-850 date which appears to be more than 50 years in the future is in fact in the past (this helps solve the "year 2000" problem).
- An HTTP/1.1 implementation may internally represent a parsed `Expires` date as earlier than the proper value, but MUST NOT internally represent a parsed `Expires` date as later than the proper value.

3.3.2 Delta Seconds

Some HTTP header fields allow a time value to be specified as an integer number of seconds, represented in decimal, after the time that the message was received. This format SHOULD only be used to represent short time periods or periods that cannot start until receipt of the message.

```
delta-seconds  = 1*DIGIT
```

3.4 Character Sets

HTTP uses the same definition of the term "character set" as that described for MIME:

The term “character set” is used in this document to refer to a method used with one or more tables to convert a sequence of octets into a sequence of characters. Note that unconditional conversion in the other direction is not required, in that not all characters may be available in a given character set and a character set may provide more than one sequence of octets to represent a particular character. This definition is intended to allow various kinds of character encodings, from simple single-table mappings such as US-ASCII to complex table switching methods such as those that use ISO 2022's techniques. However, the definition associated with a MIME character set name MUST fully specify the mapping to be performed from octets to characters. In particular, use of external profiling information to determine the exact mapping is not permitted.

Note: This use of the term “character set” is more commonly referred to as a “character encoding.” However, since HTTP and MIME share the same registry, it is important that the terminology also be shared.

HTTP character sets are identified by case-insensitive tokens. The complete set of tokens is defined by the IANA Character Set registry [19]. However, because that registry does not define a single, consistent token for each character set, we define here the preferred names for those character sets most likely to be used with HTTP entities. These character sets include those registered by RFC 1521 [7] -- the US-ASCII [21] and ISO-8859 [22] character sets -- and other names specifically recommended for use within MIME charset parameters.

```
charset = "US-ASCII"
          | "ISO-8859-1" | "ISO-8859-2" | "ISO-8859-3"
          | "ISO-8859-4" | "ISO-8859-5" | "ISO-8859-6"
          | "ISO-8859-7" | "ISO-8859-8" | "ISO-8859-9"
          | "ISO-2022-JP" | "ISO-2022-JP-2" | "ISO-2022-KR"
          | "UNICODE-1-1" | "UNICODE-1-1-UTF-7" | "UNICODE-1-1-UTF-8"
          | token
```

Although HTTP allows an arbitrary token to be used as a charset value, any token that has a predefined value within the IANA Character Set registry [19] MUST represent the character set defined by that registry. Applications SHOULD limit their use of character sets to those defined by the IANA registry.

Note: This use of the term “character set” is more commonly referred to as a “character encoding.” However, since HTTP and MIME share the same registry, it is important that the terminology also be shared.

The character set of an entity body SHOULD be labeled as the lowest common denominator of the character codes used within that body, with the exception that no label is preferred over the labels US-ASCII or ISO-8859-1.

3.5 Content Codings

Content coding values indicate an encoding transformation that has been or can be applied to a resource. Content codings are primarily used to allow a document to be compressed or encrypted without losing the identity of its underlying media type. Typically, the resource is stored in this encoding and only decoded before rendering or analogous usage.

```
content-coding   = "gzip" | "x-gzip" | "compress" | "x-compress" | token
```

Note: For historical reasons, HTTP applications SHOULD consider “x-gzip” and “x-compress” to be equivalent to “gzip” and “compress”, respectively.

All `content-coding` values are case-insensitive. HTTP/1.1 uses `content-coding` values in the `Accept-Encoding` (Section 10.3) and `Content-Encoding` (Section 10.10) header fields. Although the value describes the content-coding, what is more important is that it indicates what decoding mechanism will be required to remove the encoding. Note that a single program MAY be capable of decoding multiple content-coding formats. Two values are defined by this specification:

gzip

An encoding format produced by the file compression program "gzip" (GNU zip) developed by Jean-loup Gailly[25]. This format is typically a Lempel-Ziv coding (LZ77) with a 32 bit CRC.

compress

The encoding format produced by the file compression program "compress". This format is an adaptive Lempel-Ziv-Welch coding (LZW).

Note: Use of program names for the identification of encoding formats is not desirable and should be discouraged for future encodings. Their use here is representative of historical practice, not good design.

HTTP defines a registration process which uses the Internet Assigned Numbers Authority (IANA) as a central registry for content-coding value tokens. Additional content-coding value tokens beyond the four defined in this document (`gzip x-gzip compress x-compress`) SHOULD be registered with the IANA. To allow interoperability between clients and servers, specifications of the content coding algorithms used to implement a new value SHOULD be publicly available and adequate for independent implementation, and MUST conform to the purpose of content coding defined in this section.

3.6 Transfer Codings

Transfer coding values are used to indicate an encoding transformation that has been, can be, or may need to be applied to an `Entity-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, not of the original resource.

```
transfer-coding         = "chunked" | transfer-extension

transfer-extension      = token
```

All transfer-coding values are case-insensitive. HTTP/1.1 uses transfer coding values in the `Transfer-Encoding` header field (Section 10.39).

Transfer codings are analogous to the Content-Transfer-Encoding values of MIME [7], which were designed to enable safe transport of binary data over a 7-bit transport service. However, "safe transport" has a different focus for an 8bit-clean transfer protocol. In HTTP, the only unsafe characteristic of message bodies is the difficulty in determining the exact body length (Section 7.2.2), or the desire to encrypt data over a shared transport.

All HTTP/1.1 applications MUST be able to receive and decode the "chunked" transfer coding , and MUST ignore chunked extensions they do not understand. The chunked encoding modifies the body of a message in order to transfer it as a series of chunks, each with its own size indicator, followed by an optional footer containing entity-header fields. This allows dynamically-produced content to be transferred along with the information necessary for the recipient to verify that it has received the full message.

```
Chunked-Body   = *chunk
                 "0" CRLF
                 footer
                 CRLF

chunk          = chunk-size [ chunk-ext ] CRLF
                 chunk-data CRLF

chunk-size     = hex-no-zero *HEX
chunk-ext      = *( ";" chunk-ext-name [ "=" chunk-ext-value ] )
chunk-ext-name = token
chunk-ext-val  = token | quoted-string
chunk-data     = chunk-size(OCTET)

footer         = *< Content-MD5 and future headers that specify
                  they are allowed in footer>>

hex-no-zero    = <HEX excluding "0">
```

Note that the chunks are ended by a zero-sized chunk, followed by the footer and terminated by an empty line. An example process for decoding a `Chunked-Body` is presented in Appendix C.5.

3.7 Media Types

HTTP uses Internet Media Types [17] in the Content-Type (Section 10.15) and `Accept` (Section 10.1) header fields in order to provide open and extensible data typing and type negotiation.

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

Parameters may follow the type/subtype in the form of attribute/value pairs.

```
parameter      = attribute "=" value
attribute      = token
value          = token | quoted-string
```

The type, subtype, and parameter attribute names are case-insensitive. Parameter values may or may not be case-sensitive, depending on the semantics of the parameter name. `LWS` MUST NOT be generated between the type and subtype, nor between an attribute and its value. Upon receipt of a media type with an unrecognized parameter, a user agent SHOULD treat the media type as if the unrecognized parameter and its value were not present.

Some older HTTP applications do not recognize media type parameters. HTTP/1.1 applications SHOULD only use media type parameters when they are necessary to define the content of a message.

Media-type values are registered with the Internet Assigned Number Authority (IANA [19]). The media type registration process is outlined in RFC 1590 [17]. Use of non-registered media types is discouraged.

3.7.1 Canonicalization and Text Defaults

Internet media types are registered with a canonical form. In general, an `Entity-Body` transferred via HTTP MUST be represented in the appropriate canonical form prior to its transmission. If the body has been encoded with a `Content-Encoding`, the underlying data SHOULD be in canonical form prior to being encoded.

Media subtypes of the “text” type use `CRLF` as the text line break when in canonical form. However, HTTP allows the transport of text media with plain `CR` or `LF` alone representing a line break when used consistently within the `Entity-Body`. HTTP applications MUST accept `CRLF`, bare `CR`, and bare `LF` as being representative of a line break in text media received via HTTP.

In addition, if the text media is represented in a character set that does not use octets 13 and 10 for `CR` and `LF` respectively, as is the case for some multi-byte character sets, HTTP allows the use of whatever octet sequences are defined by that character set to represent the equivalent of `CR` and `LF` for line breaks. This flexibility regarding line breaks applies only to text media in the `Entity-Body`; a bare `CR` or `LF` SHOULD NOT be substituted for `CRLF` within any of the HTTP control structures (such as header fields and multipart boundaries).

The “charset” parameter is used with some media types to define the character set (Section 3.4) of the data. When no explicit charset parameter is provided by the sender, media subtypes of the “text” type are defined to have a default charset value of “ISO-8859-1” when received via HTTP. Data in character sets other than “ISO-8859-1” or its subsets MUST be labeled with an appropriate charset value in order to be consistently interpreted by the recipient.

> Note: Many current HTTP servers provide data using charsets other than “ISO-8859-1” without proper labeling. This situation reduces interoperability and is not recommended. To compensate for this, some HTTP user agents provide a configuration option to allow the user to change the default interpretation of the media type character set when no charset parameter is given.

3.7.2 Multipart Types

MIME provides for a number of “multipart” types -- encapsulations of one or more entities within a single message's `Entity-Body`. All multipart types share a common syntax, as defined in Section 7.2.1 of RFC 1521 [7], and MUST include a boundary parameter as part of the media type value. The message body is itself a protocol element and MUST therefore use only `CRLF` to represent line breaks between body-parts. Unlike in RFC 1521, the epilogue of any multipart message MUST be empty; HTTP applications MUST NOT transmit the epilogue even if the original resource contains an epilogue.

In HTTP, multipart body-parts MAY contain header fields which are significant to the meaning of that part.

In general, an HTTP user agent SHOULD follow the same or similar behavior as a MIME user agent would upon receipt of a multipart type. If an application receives an unrecognized multipart subtype, the application MUST treat it as being equivalent to “multipart/mixed”.

> Note: The "multipart/form-data" type has been specifically defined for carrying form data suitable for processing via the POST request method, as described in RFC 1867 [15].

3.8 Product Tokens

Product tokens are used to allow communicating applications to identify themselves via a simple product token, with an optional slash and version designator. Most fields using product tokens also allow sub-products which form a significant part of the application to be listed, separated by whitespace. By convention, the products are listed in order of their significance for identifying the application.

```
product         = token ["/" product-version]
product-version = token
```

Examples:

```
User-Agent: CERN-LineMode/2.15 libwww/2.17b3

Server: Apache/0.8.4
```

Product tokens SHOULD be short and to the point -- use of them for advertising or other non-essential information is explicitly forbidden. Although any token character may appear in a `product-version`, this token SHOULD only be used for a version identifier (i.e., successive versions of the same product SHOULD only differ in the `product-version` portion of the `product` value).

3.9 Quality Values

HTTP content negotiation (Section 12) uses short "floating point" numbers to indicate the relative importance ("weight") of various negotiable parameters. The weights are normalized to a real number in the range 0 through 1, where 0 is the minimum and 1 the maximum value. In order to discourage misuse of this feature, HTTP/1.1 applications MUST not generate more than three digits after the decimal point. User configuration of these values SHOULD also be limited in this fashion.

```
qvalue         = ( "0" [ "." 0*3DIGIT ] )
               | ( "." 0*3DIGIT )
               | ( "1" [ "." 0*3("0") ] )
```

"Quality values" is a slight misnomer, since these values actually measure relative degradation in perceived quality. Thus, a value of "0.8" represents a 20% degradation from the optimum rather than a statement of 80% quality.

3.10 Language Tags

A language tag 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` fields.

The syntax and registry of HTTP language tags is the same as that defined by RFC 1766 [1]. In summary, a language tag is composed of 1 or more parts: A primary language tag and a possibly empty series of subtags:

```
language-tag  = primary-tag *( "-" subtag )

primary-tag   = 1*8ALPHA
subtag        = 1*8ALPHA
```

Whitespace is not allowed within the tag and all tags are case-insensitive. The namespace of language tags is administered by the IANA. Example tags include:

```
en, en-US, en-cockney, i-cherokee, x-pig-latin
```

where any two-letter primary-tag is an ISO 639 language abbreviation and any two-letter initial subtag is an ISO 3166 country code. The last three tags above are not registered tags, but examples of tags which could be registered in future.

3.12 Full Date Values

Contents moved to section 3.3.

3.13 Opaque Validators

Opaque validators are quoted strings whose internal structure is not visible to clients or caches.

```
opaque-validator = strong-opaque-validator | weak-opaque-validator
                        | null-validator
strong-opaque-validator = quoted-string
weak-opaque-validator = quoted-string "/W"
null-validator = <"> <">
```

Note that the "/W" tag is considered part of a weak opaque validator; it MUST NOT be removed by any cache or client.

There are two comparison functions on opaque validators:

- The strong comparison function: in order to be considered equal, both validators must be identical in every way, and neither may be weak.
- The weak comparison function: in order to be considered equal, both validators must be identical in every way, except for the presence or absence of a "weak" tag.

The weak comparison function MAY be used for simple (non-subrange) GET requests. The strong comparison function MUST be used in all other cases.

The null validator is a special value, defined as never matching the current validator of an existing resource, and always matching the "current" validator of a resource that does not exist.

3.14 Variant IDs

Variant-IDs are used to identify specific entities (variants) of a varying resource; see section 13.8.3 for how they are used.

```
variant-id = quoted-string
```

Variant-IDs are compared using string octet-equality; case is significant.

3.15 Validator Sets

Validator sets are used for doing conditional retrievals on varying resources; see section 13.8.4.

```
validator-set = 1#validator-set-item
validator-set-item = opaque-validator
```

3.16 Variant Sets

Validator sets are used for doing conditional retrievals on varying resources; see section 13.8.3.

```
variant-set = 1#variant-set-item
variant-set-item = opaque-validator ";" variant-id
```

3.17 HTTP Protocol Parameters Related to Ranges

This section defines certain HTTP protocol parameters used in range requests and related responses.

3.17.1SLUSHY Range Units

A resource may be broken down into subranges according to various structural units.

```
range-unit = bytes-unit

other-range-unit

bytes-unit = "bytes"
```

The only range unit defined by HTTP/1.1 is “bytes”. HTTP/1.1 implementations may ignore ranges specified using other units. other-range-unit = token

3.17.2 SLUSHY Byte Ranges

Since all HTTP entities are represented in HTTP messages as sequences of bytes, the concept of a byte range is meaningful for any HTTP entity. (However, not all clients and servers need to support byte-range operations.)

Byte range specifications in HTTP apply to the sequence of bytes that would be transferred by the protocol if no transfer-encoding were being applied.

> This means that if Content-encoding is applied to the data, the byte range specification applies to the resulting content-encoded byte stream, not to the unencoded byte stream. It also means that if the entity-body's media-type is a composite type (e.g., multipart/* and message/rfc822), then the composite’s body-parts may have their own content-encoding and content-transfer-encoding, and the byte range applies to the result of the those encodings.

A byte range operation may specify a single range of bytes, or a set of ranges within a single entity.

```
ranges-specifier = byte-ranges-specifier

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.

If the last-byte-pos value is present, it must be greater than or equal to the first-byte-pos in that byte-range-spec, or the byte-range-spec is invalid. The recipient of an invalid byte-range-spec must ignore it.

If the last-byte-pos value is absent, it is assumed to be equal to the current length of the entity in bytes.

If the last-byte-pos value is larger than the current length of the entity, it is assumed to be equal to the current length of the entity.

```
suffix-byte-range-spec = "-" suffix-length

suffix-length = 1*DIGIT
```

A suffix-byte-range-spec is used to specify the suffix of the entity, of a length given by the suffix-length value. (That is, this form specifies the last N bytes of an entity.) If the entity is shorter than the specified suffix-length, the entire entity is used.

Examples of byte-ranges-specifier values (assuming an entity of length 10000):

- The first 500 bytes (byte offsets 0-499, inclusive):
  `bytes=0-499`
- The second 500 bytes (byte offsets 500-999, inclusive):
  `bytes=500-999`
- The final 500 bytes (byte offsets 9500-9999, inclusive):
  `bytes=-500`
- Or
  `bytes=9500-`
- The first and last bytes only (bytes 0 and 9999):
  `bytes=0-0,-1`
- Several legal but not canonical specifications of the second 500 bytes (byte offsets 500-999, inclusive):
  `bytes=500-600,601-999`
  `bytes=500-700,601-999`

3.17.3 SLUSHY: Content Ranges

When a server returns a partial response to a client, it must describe both the extent of the range covered by the response, and the length of the entire entity.

```
content-range-spec      = byte-content-range-spec

byte-content-range-spec = bytes-unit SP first-byte-pos "-"
                                    last-byte-pos "/" entity-length

entity-length           = 1*DIGIT
```

Unlike byte-ranges-specifier values, a byte-content-range-spec may only specify one range, and must contain absolute byte positions for both the first and last byte of the range.

A byte-content-range-spec whose last-byte-pos value, is less than its first-byte-pos value, or whose entity-length value is less than its last-byte-pos value, is invalid. The recipient of an invalid byte-content-range-spec must ignore it and any content transferred along with it.

Examples of byte-content-range-spec values, assuming that the entity contains a total of 1234 bytes:

- The first 500 bytes:
  `bytes 0-499/1234`
- The second 500 bytes:
  `bytes 500-999/1234`
- All except for the first 500 bytes:
  `bytes 500-1233/1234`

- The last 500 bytes:
  `bytes 734-1233/1234`

4. HTTP Message

4.1 Message Types

HTTP messages consist of requests from client to server and responses from server to client.

```
HTTP-message   = Full-Request                  ; HTTP/1.1 messages
               | Full-Response
               | NULL-Request
```

A NULL-Request (an empty line where a request would normally be expected) MUST be ignored. Clients SHOULD NOT send a NULL-Request, but there are some error and testing circumstances in which a NULL-Request might be sent by mistake and MUST NOT cause failure on the server.

```
NULL-Request   = CRLF
```

`Full-Request` and `Full-Response` use the generic message format of RFC 822 [9] for transferring entities. Both messages may include optional header fields (also known as "headers") and an entity body. The entity body is separated from the headers by a null line (i.e., a line with nothing preceding the `CRLF`).

```
Full-Request   = Request-Line                  ; Section 5.1
                 *( General-Header             ; Section 4.3
                  | Request-Header             ; Section 5.2
                  | Entity-Header )            ; Section 7.1
                 CRLF
                 [ Entity-Body ]               ; Section 7.2

Full-Response  = Status-Line                   ; Section 6.1
                 *( General-Header             ; Section 4.3
                  | Response-Header            ; Section 6.2
                  | Entity-Header )            ; Section 7.1
                 CRLF
                 [ Entity-Body ]               ; Section 7.2
```

4.2 Message Headers

HTTP header fields, which include `General-Header` (Section 4.3), `Request-Header` (Section 5.2), `Response-Header` (Section 6.2), and `Entity-Header` (Section 7.1) fields, follow the same generic format as that given in Section 3.1 of RFC 822 [9]. Each header field consists of a name followed by a colon (`":"`) and the field value. Field names are case-insensitive. The field value may be preceded by any amount of `LWS`, though a single `SP` is preferred. Header fields can be extended over multiple lines by preceding each extra line with at least one `SP` or `HT`.

```
HTTP-header    = field-name ":" [ field-value ] CRLF

field-name     = token
field-value    = *( field-content | LWS )

field-content  = <the OCTETs making up the field-value
                 and consisting of either *TEXT or combinations
                 of token, tspecials, and quoted-string>
```

The order in which header fields with differing field names are received is not significant. However, it is "good practice" to send `General-Header` fields first, followed by `Request-Header` or `Response-Header` fields, and ending with the Entity-Header fields.

Multiple `HTTP-header` fields with the same `field-name` may be present in a message if and only if the entire `field-value` for that header field is defined as a comma-separated list [i.e., `#(values)`]. It MUST be possible to combine the multiple header fields into one "field-name: field-value" pair, without changing the semantics of the message, by appending each subsequent field-value to the first, each separated by a comma. Thus, the order in which multiple header fields with the same field-name are received may be significant to the interpretation of the combined field-value.

4.3 General Header Fields

There are a few header fields which have general applicability for both request and response messages, but which do not apply to the entity being transferred. These headers apply only to the message being transmitted.

```
General-Header = Cache-Control            ; Section 10.8
               | Connection               ; Section 10.9
               | Date                     ; Section 10.17
               | Via                      ; Section 10.20
               | Keep-Alive               ; Section 10.24
               | Pragma                   ; Section 10.29
               | Upgrade                  ; Section 10.41
```

General header field names can be extended reliably only in combination with a change in the protocol version. However, new or experimental header fields may be given the semantics of general header fields if all parties in the communication recognize them to be general header fields. Unrecognized header fields are treated as `Entity-Header` fields.

5. Request

A request message from a client to a server includes, within the first line of that message, the method to be applied to the resource, the identifier of the resource, and the protocol version in use. For backwards compatibility with the more limited HTTP/0.9 protocol, there are two valid formats for an HTTP request:

```
       Request        = Full-Request | NULL-Request

       Full-Request   = Request-Line              ; Section 5.1
                        *( General-Header         ; Section 4.3
                         | Request-Header         ; Section 5.2
                         | Entity-Header )        ; Section 7.1
                        CRLF
                        [ Entity-Body ]           ; Section 7.2

       NULL-Request   = CRLF
```

A NULL-Request MUST be ignored.

5.1 Request-Line

The `Request-Line` begins with a method token, followed by the `Request-URI` and the protocol version, and ending with `CRLF`. The elements are separated by `SP` characters. No `CR` or `LF` are allowed except in the final `CRLF` sequence.

```
       Request-Line   = Method SP Request-URI SP HTTP-Version CRLF
```

5.1.1 Method

The `Method` token indicates the method to be performed on the resource identified by the `Request-URI`. The method is case-sensitive.

```
       Method         = "OPTIONS"                ; Section 8.1
                      | "GET"                    ; Section 8.2
                      | "HEAD"                   ; Section 8.3
                      | "POST"                   ; Section 8.4
                      | "PUT"                    ; Section 8.5
                      | "DELETE"                 ;                      | "TRACE"
; Section 8.12
                      | extension-method

       extension-method = token
```

The list of methods acceptable by a specific resource can be specified in an `Allow` header field (Section 10.5). However, the client is always notified through the return code of the response whether a method is currently allowed on a specific resource, as this can change dynamically. Servers SHOULD return the status code 405 (method not allowed) if the method is known by the server but not allowed for the requested resource, and 501 (not implemented) if the method is unrecognized or not implemented by the server. The list of methods known by a server can be listed in a Public response header field (Section 10.32).

The methods GET and HEAD MUST be supported by all general-purpose servers. Servers which provide `Last-Modified` dates for resources MUST also support the conditional GET method. All other methods are optional; however, if the above methods are implemented, they MUST be implemented with the same semantics as those specified in Section 8.

5.1.2 Request-URI

The `Request-URI` is a Uniform Resource Identifier (Section 3.2) and identifies the resource upon which to apply the request.

```
Request-URI    = "*" | absoluteURI | abs_path
```

To allow for transition to `absoluteURIs` in all requests in future versions of HTTP, HTTP/1.1 servers MUST accept the `absoluteURI` form in requests, even though HTTP/1.1 clients will not normally generate them. Versions of HTTP after HTTP/1.1 may require `absoluteURIs` everywhere, after HTTP/1.1 or later have become the dominant implementations. The three options for `Request-URI` are dependent on the nature of the request. The asterisk "*" means that the request does not apply to a particular resource, but to the server itself, and is only allowed when the Method used does not necessarily apply to a resource. One example would be

```
OPTIONS * HTTP/1.1
```

The `absoluteURI` form is only allowed to an origin server if the client knows the server supports HTTP/1.1 or later. If the `absoluteURI` form is used, any `Host` request-header included with the request MUST be ignored. The `absoluteURI` form is required when the request is being made to a proxy. The proxy is requested to forward the request and return the response. If the request is `GET` or `HEAD` and a prior response is cached, the proxy may use the cached message if it passes any restrictions in the `Cache-Control` and `Expires` header fields. Note that the proxy MAY forward the request on to another proxy or directly to the server specified by the `absoluteURI`. In order to avoid request loops, a proxy MUST be able to recognize all of its server names, including any aliases, local variations, and the numeric IP address. An example `Request-Line` would be:

```
GET http://www.w3.org/pub/WWW/TheProject.html HTTP/1.1
```

The most common form of `Request-URI` is that used to identify a resource on an origin server or gateway. In this case, only the absolute path of the URI is transmitted (see Section 3.2.1, `abs_path`). For example, a client wishing to retrieve the resource above directly from the origin server would create a TCP connection to port 80 of the host "www.w3.org" and send the lines:

```
GET /pub/WWW/TheProject.html HTTP/1.1
Host:www.w3.org
```

followed by the remainder of the `Full-Request`. Note that the absolute path cannot be empty; if none is present in the original URI, it MUST be given as "/" (the server root).

If a proxy receives a request without any path in the `Request-URI` and the method used is capable of supporting the asterisk form of request, then the last proxy on the request chain MUST forward the request with "*" as the final `Request-URI`. For example, the request

```
OPTIONS http://www.ics.uci.edu:8001 HTTP/1.1
```

would be forwarded by the proxy as

```
OPTIONS * HTTP/1.1
```

after connecting to port 8001 of host "www.ics.uci.edu".

The `Request-URI` is transmitted as an encoded string, where some characters may be escaped using the "% HEX HEX" encoding defined by RFC 1738 [4]. The origin server MUST decode the `Request-URI` in order to properly interpret the request. In requests that they forward, proxies MUST NOT rewrite the "abs_path" part of a Request-URI in any way except as noted above to replace a null abs_path with "*". Illegal Request-URIs SHOULD be responded to with an appropriate status code. (Proxies MAY transform the Request-URI for internal processing purposes, but SHOULD NOT send such a transformed Request-URI in forwarded requests. Transformations for use in cache updates and lookups are subject to additional requirements; see section 13 on caching. The main reason for this rule is to make sure that the form of Request-URIs is well specified, to enable future extensions without fear that they will break in the face of some rewritings. Another is that one consequence of rewriting the Request-URI is that integrity or authentication checks by the server may fail; since rewriting MUST be avoided in this case, it may as well be proscribed in general.

Note: servers writers SHOULD be aware that some existing proxies do some rewriting.

5.2 Request Header Fields

The request header fields allow the client to pass additional information about the request, and about the client itself, to the server. These fields act as request modifiers, with semantics equivalent to the parameters on a programming language method (procedure) invocation.

```
Request-Header = Accept                      ; Section 10.1
               | Accept-Charset              ; Section 10.2
               | Accept-Encoding             ; Section 10.3
               | Accept-Language             ; Section 10.4
               | Authorization               ; Section 10.6
               | From                        ; Section 10.21
               | Host                        ; Section 10.22
               | If-Modified-Since           ; Section 10.23
               | Proxy-Authorization         ; Section 10.31
               | Range                       ; Section 10.33
               | Referer                     ; Section 10.34
               | User-Agent                  ; Section 10.43
               | Max-Forwards                ; Section 10.45
```

Request-Header field names can be extended reliably only in combination with a change in the protocol version. However, new or experimental header fields MAY be given the semantics of request header fields if all parties in the communication recognize them to be request header fields. Unrecognized header fields are treated as Entity-Header fields.

6. Response

After receiving and interpreting a request message, a server responds in the form of an HTTP response message.

```
Response        = Full-Response

Full-Response   = Status-Line                ; Section 6.1
                  *( General-Header          ; Section 4.3
                   | Response-Header         ; Section 6.2
                   | Entity-Header )         ; Section 7.1
                  CRLF
                  [ Entity-Body ]            ; Section 7.2
```

6.1 Status-Line

The first line of a `Full-Response` message is the `Status-Line`, consisting of the protocol version followed by a numeric status code and its associated textual phrase, with each element separated by `SP` characters. No `CR` or `LF` is allowed except in the final `CRLF` sequence.

```
Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

6.1.1 Status Code and Reason Phrase

The `Status-Code` element is a 3-digit integer result code of the attempt to understand and satisfy the request. The `Reason-Phrase` is intended to give a short textual description of the `Status-Code`. The `Status-Code` is intended for use by automata and the `Reason-Phrase` is intended for the human user. The client is not required to examine or display the `Reason-Phrase`.

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:

- 1xx: Informational - Request received, continuing process
- 2xx: Success - The action was successfully received, understood, and accepted
- 3xx: Redirection - Further action must be taken in order to complete the request
- 4xx: Client Error - The request contains bad syntax or cannot be fulfilled
- 5xx: Server Error - The server failed to fulfill an apparently valid request

The individual values of the numeric status codes defined for HTTP/1.1, and an example set of corresponding `Reason-Phrase`'s, are presented below. The reason phrases listed here are only recommended -- they may be replaced by local equivalents without affecting the protocol. These codes are fully defined in Section 9.

```
Status-Code    = "100"   ; Continue
               | "101"   ; Switching Protocols
               | "200"   ; OK
               | "201"   ; Created
               | "202"   ; Accepted
               | "203"   ; Non-Authoritative Information
               | "204"   ; No Content
               | "205"   ; Reset Content
               | "206"   ; Partial Content
               | "300"   ; Multiple Choices
               | "301"   ; Moved Permanently
               | "302"   ; Moved Temporarily
               | "303"   ; See Other
               | "304"   ; Not Modified
               | "305"   ; Use Proxy
               | "400"   ; Bad Request
               | "401"   ; Unauthorized
               | "402"   ; Payment Required
               | "403"   ; Forbidden
               | "404"   ; Not Found
               | "405"   ; Method Not Allowed
               | "406"   ; Not Acceptable
               | "407"   ; Proxy Authentication Required
               | "408"   ; Request Time-out
               | "409"   ; Conflict
               | "410"   ; Gone
               | "411"   ; Length Required
               | "412"   ; Precondition Failed
               | "413"   ; Request Entity Too Large
               | "414"   ; Request URI Too Large
               | "415"   ; Unsupported Media Type
               | "416"   ; None Acceptable
               | "500"   ; Internal Server Error
               | "501"   ; Not Implemented
               | "502"   ; Bad Gateway
               | "503"   ; Service Unavailable
               | "504"   ; Gateway Time-out
               | "505"   ; HTTP Version not supported
               | extension-code

extension-code = 3DIGIT

Reason-Phrase  = *<TEXT, excluding CR, LF>
```

HTTP status codes are extensible. HTTP applications are not required to understand the meaning of all registered status codes, though such understanding is obviously desirable. However, applications MUST understand the class of any status code, as indicated by the first digit, and treat any unrecognized response as being equivalent to the x00 status code of that class, with the exception that an unrecognized response MUST not be cached. For example, if an unrecognized status code of 431 is received by the client, it can safely assume that there was something wrong with its request and treat the response as if it had received a 400 status code. In such cases, user agents SHOULD present to the user the entity returned with the response, since that entity is likely to include human-readable information which will explain the unusual status.

6.2 Response Header Fields

The response header fields allow the server to pass additional information about the response which cannot be placed in the `Status-Line`. These header fields give information about the server and about further access to the resource identified by the `Request-URI`.

```
Response-Header = Location                ; Section 10.27
                | Proxy-Authenticate      ; Section 10.30
                | Public                  ; Section 10.32
                | Retry-After             ; Section 10.36
                | Server                  ; Section 10.37
                | WWW-Authenticate        ; Section 10.44
```

`Response-Header` field names can be extended reliably only in combination with a change in the protocol version. However, new or experimental header fields MAY be given the semantics of response header fields if all parties in the communication recognize them to be response header fields. Unrecognized header fields are treated as `Entity-Header` fields.

7. Entity

`Full-Request` and `Full-Response` messages MAY transfer an entity within some requests and responses. An entity consists of `Entity-Header` fields and (usually) an `Entity-Body`. In this section, both *sender* and *recipient* refer to either the client or the server, depending on who sends and who receives the entity.

7.1 Entity Header Fields

`Entity-Header` fields define optional metainformation about the `Entity-Body` or, if no body is present, about the resource identified by the request.

```
Entity-Header  = Allow                    ; Section 10.5
               | Content-Base             ; Section 10.9
               | Content-Encoding         ; Section 10.10
               | Content-Language         ; Section 10.11
               | Content-Length           ; Section 10.12
               | Content-Location         ; Section 10.16
               | Content-MD5              ; Section 10.13
               | Content-Range            ; Section 10.14
               | Content-Type             ; Section 10.15
               | Expires                  ; Section 10.19
               | Last-Modified            ; Section 10.25
               | Title                    ; Section 10.38
               | Transfer-Encoding        ; Section 10.39
               | extension-header

extension-header = HTTP-header
```

The `extension-header` mechanism allows additional `Entity-Header` fields to be defined without changing the protocol, but these fields cannot be assumed to be recognizable by the recipient. Unrecognized header fields SHOULD be ignored by the recipient and forwarded by proxies.

7.2 Entity Body

The entity body (if any) sent with an HTTP request or response is in a format and encoding defined by the `Entity-Header` fields.

```
Entity-Body    = *OCTET
```

An entity body is included with a request message only when the request method calls for one. The presence of an entity body in a request is signaled by the inclusion of a `Content-Length` and/or `Content-Type` header field in the request message headers.

For response messages, whether or not an entity body is included with a message is dependent on both the request method and the response code. All responses to the HEAD request method MUST not include a body, even though the presence of entity header fields may lead one to believe they do. All 1xx (informational), 204 (no content), and 304 (not modified) responses MUST not include a body. All other responses MUST include an entity body or a `Content-Length` header field defined with a value of zero (0).

7.2.1 Type

When an entity body is included with a message, the data type of that body is determined via the header fields `Content-Type`, `Content-Encoding`, and `Transfer-Encoding`. These define a three-layer, ordered encoding model:

```
entity-body :=
   Transfer-Encoding( Content-Encoding( Content-Type( data ) ) )
```

The default for both encodings is none (i.e., the identity function). `Content-Type` specifies the media type of the underlying data. `Content-Encoding` may be used to indicate any additional content codings applied to the type, usually for the purpose of data compression, that are a property of the resource requested. `Transfer-Encoding` may be used to indicate any additional transfer codings applied by an application to ensure safe and proper transfer of the message. Note that `Transfer-Encoding` is a property of the message, not of the resource.

Any HTTP/1.1 message containing an entity body SHOULD include a `Content-Type` header field defining the media type of that body. If and **only if** the media type is not given by a `Content-Type` header, the recipient may attempt to guess the media type via inspection of its content and/or the name extension(s) of the URL used to identify the resource. If the media type remains unknown, the recipient SHOULD treat it as type “`application/octet-stream`”.

7.2.2 Length

When an entity body is included with a message, the length of that body may be determined in one of several ways. If a `Content-Length` header field is present, its value in bytes represents the length of the entity body. Otherwise, the body length is determined by the `Transfer-Encoding` (if the “chunked” transfer coding has been applied) or by the server closing the connection.

> Note: Any response message which MUST NOT include an entity body (such as the 1xx, 204, and 304 responses and any response to a HEAD request) is always terminated by the first empty line after the header fields, regardless of the entity header fields present in the message.

Closing the connection cannot be used to indicate the end of a request body, since it leaves no possibility for the server to send back a response. For compatibility with HTTP/1.0 applications, HTTP/1.1 requests containing an entity body MUST include a valid `Content-Length` header field unless the server is known to be HTTP/1.1 compliant. HTTP/1.1 servers MUST accept the "chunked" transfer coding (Section 3.6), thus allowing this mechanism to be used for a request when `Content-Length` is unknown.

If a request contains an entity body and `Content-Length` is not specified, the server SHOULD respond with 400 (bad request) if it cannot determine the length of the request message's content, or with 411 (length required) if it wishes to insist on receiving a valid `Content-Length`.

Messages MUST NOT include both a `Content-Length` header field and the "chunked" transfer coding. If both are received, the `Content-Length` MUST be ignored.

When a `Content-Length` is given in a message where an entity body is allowed, its field value MUST exactly match the number of `OCTETs` in the entity body. HTTP/1.1 user agents MUST notify the user when an invalid length is received and detected.

8. Method Definitions

The set of common methods for HTTP/1.1 is defined below. Although this set can be expanded, additional methods cannot be assumed to share the same semantics for separately extended clients and servers.

The `Host` request-header field (Section 10.22) MUST accompany all HTTP/1.1 requests.

8.1 OPTIONS

The `OPTIONS` method represents a request for information about the communication options available on the request/response chain identified by the `Request-URI`. This method allows the client to determine the options and/or requirements associated with a resource, or the capabilities of a server, without implying a resource action or initiating a resource retrieval.

Unless the server's response is an error, the response MUST NOT include entity information other than what can be considered as communication options (e.g., `Allow` is appropriate, but `Content-Type` is not) and MUST include a `Content-Length` with a value of zero (0). Responses to this method are not cachable.

If the `Request-URI` is an asterisk ("*"), the `OPTIONS` request is intended to apply to the server as a whole. A 200 response SHOULD include any header fields which indicate optional features implemented by the server (e.g., `Public`), including any extensions not defined by this specification, in addition to any applicable general or response header fields. As described in Section 5.1.2, an "`OPTIONS *`" request can be applied through a proxy by specifying the destination server in the `Request-URI` without any path information.

If the `Request-URI` is not an asterisk, the `OPTIONS` request applies only to the options that are available when communicating with that resource. A 200 response SHOULD include any header fields which indicate optional features implemented by the server and applicable to that resource (e.g., `Allow`), including any extensions not defined by this specification, in addition to any applicable general or response header fields. If the `OPTIONS` request passes through a proxy, the proxy MUST edit the response to exclude those options known to be unavailable through that proxy.

8.2 GET

The `GET` method means retrieve whatever information (in the form of an entity) is identified by the `Request-URI`. If the `Request-URI` refers to a data-producing process, it is the produced data which shall be returned as the entity in the response and not the source text of the process, unless that text happens to be the output of the process.

The semantics of the `GET` method change to a "conditional `GET`" if the request message includes an `If-Modified-Since` header field. A conditional `GET` method requests that the identified resource be transferred only if it has been modified since the date given by the `If-Modified-Since` header, as described in Section 10.23. The conditional `GET` method is intended to reduce unnecessary network usage by allowing cached entities to be refreshed without requiring multiple requests or transferring data already held by the client.

The semantics of the `GET` method change to a “partial `GET`” if the request message includes a `Range` header field. A partial `GET` requests that only part of the identified resource be transferred, as described in Section 10.33. The partial `GET` method is intended to reduce unnecessary network usage by allowing partially-retrieved entities to be completed without transferring data already held by the client.

The response to a `GET` request may be cachable if and only if it meets the requirements for HTTP caching described in Section 13.

8.3 HEAD

The `HEAD` method is identical to `GET` except that the server MUST not return any `Entity-Body` in the response. The metainformation contained in the HTTP headers in response to a `HEAD` request SHOULD be identical to the information sent in response to a `GET` request. This method can be used for obtaining metainformation about the resource identified by the `Request-URI` without transferring the `Entity-Body` itself. This method is often used for testing hypertext links for validity, accessibility, and recent modification.

The response to a `HEAD` request may be cachable in the sense that the information contained in the response may be used to update a previously cached entity from that resource. If the new field values indicate that the cached entity differs from the current resource (as would be indicated by a change in `Content-Length`, `Content-MD5`, or `Content-Version`), then the cache MUST discard the cached entity.

There is no “conditional `HEAD`” or “partial `HEAD`” request analogous to those associated with the `GET` method. If an `If-Modified-Since` and/or `Range` header field is included with a `HEAD` request, they SHOULD be ignored.

8.4 POST

The `POST` method is used to request that the destination server accept the entity enclosed in the request as a new subordinate of the resource identified by the `Request-URI` in the `Request-Line`. POST is designed to allow a uniform method to cover the following functions:

- Annotation of existing resources;
- Posting a message to a bulletin board, newsgroup, mailing list, or similar group of articles;
- Providing a block of data, such as the result of submitting a form [5], to a data-handling process;
- Extending a database through an append operation.

The actual function performed by the POST method is determined by the server and is usually dependent on the `Request-URI`. The posted entity is subordinate to that URI in the same way that a file is subordinate to a directory containing it, a news article is subordinate to a newsgroup to which it is posted, or a record is subordinate to a database.

For compatibility with HTTP/1.0 applications, all `POST` requests MUST include a valid `Content-Length` header field unless the server is known to be HTTP/1.1 compliant. When sending a `POST` request to an HTTP/1.1 server, a client MUST use a valid `Content-Length` or the “chunked” `Transfer-Encoding`. The server SHOULD respond with a 400 (bad request) message if it cannot determine the length of the request message's content, or with 411 (length required) if it wishes to insist on receiving a valid `Content-Length`.

A successful `POST` does not require that the entity be created as a resource on the origin server or made accessible for future reference. That is, the action performed by the `POST` method might not result in a resource that can be identified by a URI. In this case, either 200 (ok) or 204 (no content) is the appropriate response status, depending on whether or not the response includes an entity that describes the result.

If a resource has been created on the origin server, the response SHOULD be 201 (created) and contain an entity (preferably of type "text/html") which describes the status of the request and refers to the new resource.

Responses to this method are not cachable. However, the 303 (see other) response can be used to direct the user agent to retrieve a cachable resource.

POST requests must obey the entity transmission requirements set out in section 8.4.1.

8.4.1 SLUSHY: Entity Transmission Requirements

The following rules apply to any method that is subject to the two-phase mechanism.

Upon receiving such a method from an HTTP/1.1 (or later) client, an HTTP/1.1 (or later) server immediately either respond with "100 Continue" and continue to read from the input stream, or respond with an error status. If it responds with an error status, it MAY close the transport (TCP) connection or it MAY continue to read and discard the rest of the request. It MUST not perform the requested action if returns an error status.

HTTP/1.1 servers are encouraged to maintain persistent connections and use TCP's flow control mechanisms to resolve temporary overloads, rather than terminating connections with the expectation that clients will retry. The latter technique can exacerbate network congestion.

An HTTP/1.1 (or later) client doing a PUT-like method SHOULD monitor the network connection for an error status while it is transmitting the body of the request including any encoding mechanism used to transmit the body. If the client sees an error status, it SHOULD immediately cease transmitting the body. If the body was proceeded by a `Content-length` header, the client MUST either close the connection or if the body is being sent using a Chunked encoding, use a 0 length chunk, to mark the end of the message.

An HTTP/1.1 (or later) client MUST be prepared to accept a 100 Continue status followed by a regular response.

An HTTP/1.1 (or later) client that sees the connection close before receiving any status from the server SHOULD retry the request, but if it does so, it MUST use the two-phase mechanism. In the two-phase mechanism, the client first sends the request headers, then waits for the server to respond with either a 100 Continue, in which case the client SHOULD continue, or an error status, in which case the client MUST NOT continue and MUST close the connection if it has not already completed sending the full request body including any encoding mechanism used to transmit the body.

If the client knows that the server is an HTTP/1.1 (or later) server, because of the server protocol version returned with a previous request on the same persistent connection **[alternatively: within the past <N> hours],** it MUST wait for a response. If the client believes that the server is a 1.0 or earlier server, it SHOULD continue transmitting its request after waiting at least [5] seconds for a status response.

An HTTP/1.1 (or later) client that sees the connection close after receiving a "100 Continue" but before receiving any other status SHOULD retry the request, and need not use the two-phase method (but MAY do so if this simplifies the implementation).

An HTTP/1.1 (or later) server that receives a request from a 1.0 (or earlier) client MUST NOT transmit the "100 Continue" response; it SHOULD either wait for the request to be completed normally (thus avoiding an interrupted request) or close the connection prematurely.

8.5 PUT

The PUT method requests that the enclosed entity be stored under the supplied Request-URI. If the Request-URI refers to an already existing resource, the enclosed entity SHOULD be considered as a modified version of the one residing on the origin server. If the Request-URI does not point to an existing resource, and that URI is capable of being defined as a new resource by the requesting user agent, the origin server can create the resource with that URI. If a new resource is created, the origin server MUST inform the user agent via the 201 (created) response. If an existing resource is modified, either the 200 (ok) or 204 (no content) response codes SHOULD be sent to indicate successful completion of the request. If the resource could not be created or modified with the Request-URI, an appropriate error response SHOULD be given that reflects the nature of the problem.

If the request passes through a cache and the `Request-URI` identifies a currently cached entity, that entity MUST be removed from the cache. Responses to this method are not cachable.

The fundamental difference between the `POST` and `PUT` requests is reflected in the different meaning of the `Request-URI`. The URI in a `POST` request identifies the resource that will handle the enclosed entity as an appendage. That resource may be a data-accepting process, a gateway to some other protocol, or a separate entity that accepts annotations. In contrast, the URI in a `PUT` request identifies the entity enclosed with the request -- the user agent knows what URI is intended and the server MUST NOT attempt to apply the request to some other resource. If the server desires that the request be applied to a different URI, it MUST send a 301 (moved permanently) response; the user agent MAY then make its own decision regarding whether or not to redirect the request.

A single resource MAY be identified by many different URIs. For example, an article may have a URI for identifying “the current version” which is separate from the URI identifying each particular version. In this case, a `PUT` request on a general URI may result in several other URIs being defined by the origin server.

For compatibility with HTTP/1.0 applications, all `PUT` requests MUST include a valid `Content-Length` header field unless the server is known to be HTTP/1.1 compliant. When sending a `PUT` request to an HTTP/1.1 server, a client MUST use a valid `Content-Length` or the “chunked” `Transfer-Encoding`. The server SHOULD respond with a 400 (bad request) message if it cannot determine the length of the request message's content, or with 411 (length required) if it wishes to insist on receiving a valid `Content-Length`.

The actual method for determining how the resource is placed, and what happens to its predecessor, is defined entirely by the origin server. If the entity being `PUT` was derived from an existing resource which included a `Content-Version` header field, the new entity MUST include a `Derived-From` header field corresponding to the value of the original `Content-Version` header field. Multiple `Derived-From` values may be included if the entity was derived from multiple resources with `Content-Version` information. Applications are encouraged to use these fields for constructing versioning relationships and resolving version conflicts.

PUT requests must obey the entity transmission requirements set out in section 8.4.1.

8.9 DELETE

The DELETE method requests that the origin server delete the resource identified by the `Request-URI`. This method MAY be overridden by human intervention (or other means) on the origin server. The client cannot be guaranteed that the operation has been carried out, even if the status code returned from the origin server indicates that the action has been completed successfully. However, the server SHOULD not indicate success unless, at the time the response is given, it intends to delete the resource or move it to an inaccessible location.

A successful response SHOULD be 200 (OK) if the response includes an entity describing the status, 202 (accepted) if the action has not yet been enacted, or 204 (no content) if the response is OK but does not include an entity.

If the request passes through a cache and the `Request-URI` identifies a currently cached entity, that entity MUST be removed from the cache. Responses to this method are not cachable.

8.12 TRACE

The TRACE method is used to invoke a remote, application-layer loop back of the request message. The final recipient of the request SHOULD reflect the message received back to the client as the entity body of a 200 (OK) response. The final recipient is either the origin server or the first proxy or gateway to receive a `Max-Forwards` value of zero (0) in the request (see Section 10.45). A TRACE request MUST NOT include an entity body and MUST include a `Content-Length` header field with a value of zero (0).

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 10.20) 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.

If successful, the response SHOULD contain the entire request message in the entity body, with a `Content-Type` of “message/http”, “application/http”, or “text/plain”. Responses to this method MUST NOT be cached.

9. Status Code Definitions

Each `Status-Code` is described below, including a description of which `method`(s) it can follow and any metainformation required in the response.

9.1 Informational 1xx

This class of status code indicates a provisional response, consisting only of the `Status-Line` and optional headers, and is terminated by an empty line. Since HTTP/1.0 did not define any 1xx status codes, servers MUST NOT send a 1xx response to an HTTP/1.0 client except under experimental conditions.

100 Continue

The client may continue with its request. This interim response is used to inform the client that the initial part of the request has been received and has not yet been rejected by the server. The client SHOULD continue by sending the remainder of the request or, if the request has already been completed, ignore this response. The server MUST send a final response after the request has been completed.

101 Switching Protocols

The server understands and is willing to comply with the client's request, via the `Upgrade` message header field (Section 10.41), for a change in the application protocol being used on this connection. The server will switch protocols to those defined by the response's `Upgrade` header field immediately after the empty line which terminates the 101 response.

The protocol should only be switched when it is advantageous to do so. For example, switching to a newer version of HTTP is advantageous over older versions, and switching to a real-time, synchronous protocol may be advantageous when delivering resources that use such features.

9.2 Successful 2xx

This class of status code indicates that the client's request was successfully received, understood, and accepted.

200 OK

The request has succeeded. The information returned with the response is dependent on the method used in the request, as follows:

`GET`
: an entity corresponding to the requested resource is sent in the response;

`HEAD`
: the response MUST only contain the header information and no `Entity-Body`;

`POST`
: an entity describing or containing the result of the action;

`TRACE`
: an entity containing the request message as received by the end server;

otherwise,
: an entity describing the result of the action;

If the entity corresponds to a resource, the response MAY include a `Content-Location` header field giving the actual location of that specific resource for later reference.

201 Created

The request has been fulfilled and resulted in a new resource being created. The newly created resource can be referenced by the URI(s) returned in the entity of the response, with the most specific URL for the resource given by a `Location` header field. The origin server SHOULD create the resource before using this `Status-Code`. If the action cannot be carried out immediately, the server MUST include in the response body a description of when the resource will be available; otherwise, the server SHOULD respond with 202 (accepted).

202 Accepted

The request has been accepted for processing, but the processing has not been completed. The request MAY or MAY NOT eventually be acted upon, as it MAY be disallowed when processing actually takes place. There is no facility for re-sending a status code from an asynchronous operation such as this.

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 entity returned with this response SHOULD include an indication of the request's current status and either a pointer to a status monitor or some estimate of when the user can expect the request to be fulfilled.

203 Non-Authoritative Information

The returned metainformation in the Entity-Header is not the definitive set as available from the origin server, but is gathered from a local or a third-party copy. The set presented MAY be a subset or superset of the original version. For example, including local annotation information about the resource MAY result in a superset of the metainformation known by the origin server. Use of this response code is not required and is only appropriate when the response would otherwise be 200 (OK).

204 No Content

The server has fulfilled the request but there is no new information to send back. If the client is a user agent, it SHOULD not change its document view from that which caused the request to be generated. This response is primarily intended to allow input for actions to take place without causing a change to the user agent's active document view. The response MAY include new metainformation in the form of entity headers, which SHOULD apply to the document currently in the user agent's active view.

The 204 response MUST not include an entity body, and thus is always terminated by the first empty line after the header fields.

205 Reset Content

The server has fulfilled the request and the user agent SHOULD reset the document view which caused the request to be generated. This response is primarily intended to allow input for actions to take place via user input, followed by a clearing of the form in which the input is given so that the user can easily initiate another input action. The response MUST include a `Content-Length` with a value of zero (0) and no entity body.

206 Partial Content

The server has fulfilled the partial `GET` request for the resource. The request MUST have included a `Range` header field (Section 10.33) indicating the desired range. The response MUST include a `Content-Range` header field (Section 10.14) indicating the range included with this response. All entity header fields in the response MUST describe the partial entity transmitted rather than what would have been transmitted in a full response. In particular, the `Content-Length` header field in the response MUST match the actual number of `OCTETs` transmitted in the entity body. It is assumed that the client already has the complete entity's header field data.

207 Range Out Of Bounds

The server has determined that the requested range(s) are not present in the requested resource, and so there is no content to return. This status code should be handled by the client the same as 204 No Content.

> **This could be a compatibility problem if there is an installed base. If treating this status code as the generic 2xx code by such implementations would lead to an error, it will have to be replace by 204.**

9.3 Redirection 3xx

This class of status code indicates that further action needs to be taken by the user agent in order to fulfill the request. The action required MAY be carried out by the user agent without interaction with the user if and only if the method used in the second request is `GET` or `HEAD`. A user agent SHOULD NOT automatically redirect a request more than 5 times, since such redirections usually indicate an infinite loop.

300 Multiple Choices

This status code is reserved for future use by a planned content negotiation mechanism. HTTP/1.1 user agents receiving a 300 response which includes a `Location` header field can treat this response as they would treat a 303 (See Other) response. If no `Location` header field is included, the appropriate action is to display the entity enclosed in the response to the user.

301 Moved Permanently

The requested resource has been assigned a new permanent URI and any future references to this resource SHOULD be done using one of the returned URIs. Clients with link editing capabilities SHOULD automatically re-link references to the `Request-URI` to one or more of the new references returned by the server, where possible. This response is cachable unless indicated otherwise.

If the new URI is a location, its URL MUST be given by the `Location` field in the response. Unless it was a `HEAD` request, the `Entity-Body` of the response SHOULD contain a short hypertext note with a hyperlink to the new URI(s).

If the 301 status code is received in response to a request other than GET or HEAD, the user agent MUST NOT automatically redirect the request unless it can be confirmed by the user, since this might change the conditions under which the request was issued.

> Note: When automatically redirecting a POST request after receiving a 301 status code, some existing HTTP/1.0 user agents will erroneously change it into a GET request.

302 Moved Temporarily

The requested resource resides temporarily under a different URI. Since the redirection MAY be altered on occasion, the client SHOULD continue to use the Request-URI for future requests. This response is only cachable if indicated by a Cache-Control or Expires header field.

If the new URI is a location, its URL MUST be given by the Location field in the response. Unless it was a HEAD request, the Entity-Body of the response SHOULD contain a short hypertext note with a hyperlink to the new URI(s).

If the 302 status code is received in response to a request other than GET or HEAD, the user agent MUST NOT automatically redirect the request unless it can be confirmed by the user, since this might change the conditions under which the request was issued.

303 See Other

The response to the request can be found under a different URI and SHOULD be retrieved using a GET method on that resource. This method exists primarily to allow the output of a POST-activated script to redirect the user agent to a selected resource. The new resource is not a update reference for the original Request-URI. The 303 response is not cachable, but the response to the second request MAY be cachable.

If the new URI is a location, its URL MUST be given by the Location field in the response. Unless it was a HEAD request, the Entity-Body of the response SHOULD contain a short hypertext note with a hyperlink to the new URI(s).

> Note: When automatically redirecting a POST request after receiving a 302 status code, some existing HTTP/1.0 user agents will erroneously change it into a GET request.

304 Not Modified

If the client has performed a conditional GET request and access is allowed, but the document has not been modified since the date and time specified in the If-Modified-Since field, the server MUST respond with this status code and not send an Entity-Body to the client. Header fields contained in the response SHOULD only include information which is relevant to cache managers or which MAY have changed independently of the entity's Last-Modified date. Examples of relevant header fields include: Date, Server, Content-Length, Content-MD5, Content-Version, Cache-Control and Expires.

A cache SHOULD update its cached entity to reflect any new field values given in the 304 response. If the new field values indicate that the cached entity differs from the current resource (as would be indicated by a change in Content-Length, Content-MD5, or Content-Version), then the cache MUST disregard the 304 response and repeat the request without an If-Modified-Since field.

The 304 response MUST NOT include an entity body, and thus is always terminated by the first empty line after the header fields.

305 Use Proxy

The requested resource MUST be accessed through the proxy given by the Location field in the response. In other words, this is a proxy redirect.

9.4 Client Error 4xx

The 4xx class of status code is intended for cases in which the client seems to have erred. If the client has not completed the request when a 4xx code is received, it SHOULD immediately cease sending data to the server. Except when responding to a HEAD request, the server SHOULD include an entity 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.

> Note: If the client is sending data, server implementations on TCP SHOULD be careful to ensure that the client acknowledges receipt of the packet(s) containing the response prior to closing the input connection. If the client continues sending data to the server after the close, the server's controller will send a reset packet to the client, which may erase the client's unacknowledged input buffers before they can be read and interpreted by the HTTP application.

400 Bad Request

The request could not be understood by the server due to malformed syntax. The client SHOULD not repeat the request without modifications.

401 Unauthorized

The request requires user authentication. The response MUST include a `WWW-Authenticate` header field (Section 10.44) containing a `challenge` applicable to the requested resource. The client MAY repeat the request with a suitable `Authorization` header field (Section 10.6). If the request already included Authorization credentials, then the 401 response indicates that authorization has been refused for those credentials. 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 SHOULD be presented the entity that was given in the response, since that entity MAY include relevant diagnostic information. HTTP access authentication is explained in Section 11.

402 Payment Required

This code is reserved for future use.

403 Forbidden

The server understood the request, but is refusing to fulfill it. Authorization will not help and the request SHOULD not be repeated. If the request method was not `HEAD` and the server wishes to make public why the request has not been fulfilled, it SHOULD describe the reason for the refusal in the entity body. This status code is commonly used when the server does not wish to reveal exactly why the request has been refused, or when no other response is applicable.

404 Not Found

The server has not found anything matching the `Request-URI`. No indication is given of whether the condition is temporary or permanent. If the server does not wish to make this information available to the client, the status code 403 (forbidden) can be used instead. The 410 (gone) status code SHOULD be used if the server knows, through some internally configurable mechanism, that an old resource is permanently unavailable and has no forwarding address.

405 Method Not Allowed

The method specified in the `Request-Line` is not allowed for the resource identified by the `Request-URI`. The response MUST include an Allow header containing a list of valid methods for the requested resource.

406 Not Acceptable

The resource identified by the request is only capable of generating response entities which have content characteristics not acceptable according to the accept headers sent in the request.

HTTP/1.1 servers are allowed to return responses which are not acceptable according to the accept headers sent in the request. In some cases, this may even be preferable over sending a 406 response. User agents are encouraged to inspect the headers of an incoming response to determine if it is acceptable. If the response is not acceptable, user agents SHOULD interrupt the receipt of the response if doing so would save network resources. If it is unknown whether an incoming response would be acceptable, a user agent SHOULD temporarily stop receipt of more data and query the user for a decision on further actions.

407 Proxy Authentication Required

This code is similar to 401 (unauthorized), but indicates that the client MUST first authenticate itself with the proxy. The proxy MUST return a `Proxy-Authenticate` header field (Section 10.30) containing a challenge applicable to the proxy for the requested resource. The client MAY repeat the request with a suitable `Proxy-Authorization` header field (Section 10.31). HTTP access authentication is explained in Section 11.

408 Request Timeout

The client did not produce a request within the time that the server was prepared to wait. The client MAY repeat the request without modifications at any later time.

409 Conflict

The request could not be completed due to a conflict with the current state of the resource. This code is only allowed in situations where it is expected that the user MAY be able to resolve the conflict and resubmit the request. The response body SHOULD include enough information for the user to recognize the source of the conflict. Ideally, the response entity would include enough information for the user or user-agent to fix the problem; however, that MAY not be possible and is not required.

Conflicts are most likely to occur in response to a `PUT` request. If versioning is being used and the entity being `PUT` includes changes to a resource which conflict with those made by an earlier (third-party) request, the server MAY use the 409 response to indicate that it can't complete the request. In this case, the response entity SHOULD contain a list of the differences between the two versions in a format defined by the response `Content-Type`.

410 Gone

The requested resource is no longer available at the server and no forwarding address is known. This condition SHOULD be considered permanent. Clients with link editing capabilities SHOULD delete references to the `Request-URI` after user approval. If the server does not know, or has no facility to determine, whether or not the condition is permanent, the status code 404 (not found) SHOULD be used instead. This response is cachable unless indicated otherwise.

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 working at the 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.

411 Length Required

The server refuses to accept the request without a defined `Content-Length`. The client MAY repeat the request if it adds a valid `Content-Length` header field containing the length of the entity body in the request message.

412 Precondition Failed

The precondition given in one or more of the request header fields evaluated to false when it was tested on the server. This response code allows the client to place preconditions on the current resource metainformation (header field data) and thus prevent the requested method from being applied to a resource other than the one intended.

413 Request Entity Too Large

The server is refusing to process a request because it considers the request entity to be larger than it is willing or able to process. The server SHOULD close the connection if that is necessary to prevent the client from continuing the request.

If the client manages to read the 413 response, it MUST honor it and SHOULD reflect it to the user.

If this restriction is considered temporary, the server MAY include a `Retry-After` header field to indicate that it is temporary and after what time the client MAY try again.

414 Request-URI Too Large

The server is refusing to service the request because the `Request-URI` 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 URL "black hole" of redirection (e.g., a redirected URL prefix that points to a suffix of itself), or when the server is under attack by a client attempting to exploit security holes present in some servers using fixed-length buffers for reading or manipulating the `Request-URI`.

415 Unsupported Media Type

The server is refusing to service the request because the entity body of the request is in a format not supported by the requested resource for the requested method.

416 None Acceptable

This status code is reserved for future use by a planned content negotiation mechanism. HTTP/1.1 user agents receiving a 416 response which includes a `Location` header can treat this response as they would treat a 303 (See Other) response. If no `Location` header is included, the appropriate action is to display the entity enclosed in the response to the user.

9.5 Server Error 5xx

Response status codes beginning with the digit "5" indicate cases in which the server is aware that it has erred or is incapable of performing the request. If the client has not completed the request when a 5xx code is received, it SHOULD immediately cease sending data to the server. Except when responding to a HEAD request, the server SHOULD include an entity containing an explanation of the error situation, and whether it is a temporary or permanent condition. These response codes are applicable to any request method and there are no required header fields.

500 Internal Server Error

The server encountered an unexpected condition which prevented it from fulfilling the request.

501 Not Implemented

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.

502 Bad Gateway

The server, while acting as a gateway or proxy, received an invalid response from the upstream server it accessed in attempting to fulfill the request.

503 Service Unavailable

The server is currently unable to handle the request due to a temporary overloading or maintenance of the server. The implication is that this is a temporary condition which will be alleviated after some delay. If known, the length of the delay MAY be indicated in a `Retry-After` header. If no `Retry-After` is given, the client SHOULD handle the response as it would for a 500 response.

> Note: The existence of the 503 status code does not imply that a server must use it when becoming overloaded. Some servers MAY wish to simply refuse the connection.

504 Gateway Timeout

The server, while acting as a gateway or proxy, did not receive a timely response from the upstream server it accessed in attempting to complete the request.

505 HTTP Version Not Supported

The server does not support, or refuses to support, the HTTP protocol version 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 3.1, other than with this error message. The response SHOULD contain an entity describing why that version is not supported and what other protocols are supported by that server.

10. Header Field Definitions

This section defines the syntax and semantics of all standard HTTP/1.1 header fields. For `Entity-Header` fields, both *sender* and *recipient* refer to either the client or the server, depending on who sends and who receives the entity.

10.1 Accept

The `Accept` request-header field can be used to specify certain media types which are acceptable for the response. `Accept` headers 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.

The field MAY be folded onto several lines and more than one occurrence of the field is allowed, with the semantics being the same as if all the entries had been in one field value.

```
Accept         = "Accept" ":" #(
                       media-range
                       [ ( ":" | ";" )
                         range-parameter
                         *( ";" range-parameter ) ]
                      | extension-token )


media-range    = ( "*/*"
                 | ( type "/" "*" )
                 | ( type "/" subtype )
                 ) *( ";" parameter )

range-parameter = ( "q" "=" qvalue )
                | extension-range-parameter

extension-range-parameter = ( token "=" token )

extension-token = token
```

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 range-parameter q is used to indicate the media type quality factor for the range, which represents the user's preference for that range of media types. The default value is q=1. In `Accept` headers generated by HTTP/1.1 clients, the character separating media-ranges from range-parameters SHOULD be a ":". HTTP/1.1 servers SHOULD be tolerant of use of the ";" separator by HTTP/1.0 clients.

The example

```
Accept: audio/*: q=0.2, audio/basic
```

SHOULD be 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."

If no `Accept` header is present, then it is assumed that the client accepts all media types. If `Accept` headers are present, and if the server cannot send a response which is acceptable according to the `Accept` headers, then the server SHOULD send an error response with the 406 (not acceptable) status code, though the sending of an unacceptable response is also allowed.

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 preferred media types, but if they do not exist, then send the text/x-dvi entity, and if that does not exist, send the text/plain entity."

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/html, text/html;level=1, */*
```

have the following precedence:

```
1) text/html;level=1
2) text/html
3) text/*
4) */*
```

The media type quality factor associated with a given type is determined by finding the media range with the highest precedence which matches that type. For example,

```
Accept: text/*:q=0.3, text/html:q=0.7, text/html;level=1,
        */*:q=0.5
```

would cause the following values to be associated:

```
text/html;level=1                  = 1
text/html                                  = 0.7
text/plain                                 = 0.3
image/jpeg                                 = 0.5
text/html;level=3                          = 0.7
```

Note: A user agent MAY be provided with a default set of quality values for certain media ranges. However, unless the user agent is a closed system which cannot interact with other rendering agents, this default set SHOULD be configurable by the user.

10.2 Accept-Charset

The `Accept-Charset` request-header field can be used to indicate what character sets are acceptable for the response. This field allows clients capable of understanding more comprehensive or special-purpose character sets to signal that capability to a server which is capable of representing documents in those character sets. The ISO-8859-1 character set can be assumed to be acceptable to all user agents.

```
Accept-Charset = "Accept-Charset" ":"

          1#( charset [ ";" "q" "=" qvalue ] )
```

Character set values are described in Section 3.4. Each charset may be given an associated quality value which represents the user's preference for that charset. The default value is q=1. An example is

```
Accept-Charset: iso-8859-5, unicode-1-1;q=0.8
```

If no `Accept-Charset` header is present, the default is that any character set is acceptable. If an `Accept-Charset` header is present, and if the server cannot send a response which is acceptable according to the `Accept-Charset` header, then the server SHOULD send an error response with the 406 (not acceptable) status code, though the sending of an unacceptable response is also allowed.

10.3 Accept-Encoding

The `Accept-Encoding` request-header field is similar to `Accept`, but restricts the content-coding values (Section 3.5) which are acceptable in the response.

```
Accept-Encoding         = "Accept-Encoding" ":"
                          #( content-coding )
```

An example of its use is

```
Accept-Encoding: compress, gzip
```

If no `Accept-Encoding` header is present in a request, the server MAY assume that the client will accept any content coding. If an `Accept-Encoding` header is present, and if the server cannot send a response which is acceptable according to the `Accept-Encoding` header, then the server SHOULD send an error response with the 406 (not acceptable) status code.

10.4 Accept-Language

The `Accept-Language` request-header field is similar to `Accept`, but restricts the set of natural languages that are preferred as a response to the request.

```
Accept-Language         = "Accept-Language" ":"
                          1#( language-range [ ";" "q" "=" qvalue ] )

language-range           = ( ( 1*8ALPHA *( "-" 1*8ALPHA ) )
                  | "*" )
```

Each language-range MAY be given an associated quality value which represents an estimate of the user's comprehension of the languages specified by that range. The quality value defaults to “q=1” (100% comprehension).For example,

```
Accept-Language: da, en-gb;q=0.8, en;q=0.7
```

would mean: “I prefer Danish, but will accept British English (with 80% comprehension) and other types of English(with 70% comprehension).” A language-range matches a language-tag if it exactly equals the tag, or if it exactly equals a prefix (a sub-sequence starting at the first character) of the tag such that the first tag character following the prefix is “-”. The special range “*”, if present in the `Accept-Language` field, matches every tag not matched by any other ranges present in the `Accept-Language` field.

> Note: This use of a prefix matching rule does not imply that language tags are assigned to languages in such a way that it is always true that if a user understands a language with a certain tag, then this user will also understand all languages with tags for which this tag is a prefix. The prefix rule simply allows the use of prefix tags if this is the case.

The language quality factor assigned to a language-tag by the `Accept-Language` field is the quality value of the longest `language-range` in the field that matches the `language-range`. If no `language-range` in the field matches the tag, the language quality factor assigned is 0. If no `Accept-Language` header is present in the request, the server SHOULD assume that all languages are equally acceptable. If an `Accept-Language` header is present, then all languages which are assigned a quality factor greater than 0 are acceptable. If the server cannot generate a response for an audience capable of understanding at least one acceptable language, it can send a response that uses one or more un-accepted languages.

It may be contrary to the privacy expectations of the user to send an `Accept-Language` header with the complete linguistic preferences of the user in every request. For a discussion of this issue, see Section 14.7.

> Note: As intelligibility is highly dependent on the individual user, it is recommended that client applications make the choice of linguistic preference available to the user. If the choice is not made available, then the `Accept-Language` header field MUST not be given in the request.

10.5 Allow

The `Allow` entity-header field lists the set of methods supported by the resource identified by the `Request-URI`. The purpose of this field is strictly to inform the recipient of valid methods associated with the resource. An `Allow` header field MUST be present in a 405 (method not allowed) response. The `Allow` header field is not permitted in a request using the POST method, and thus SHOULD be ignored if it is received as part of a POST entity.

```
Allow          = "Allow" ":" 1#method
```

Example of use:

```
Allow: GET, HEAD, PUT
```

This field cannot prevent a client from trying other methods. However, the indications given by the `Allow` header field value SHOULD be followed. The actual set of allowed methods is defined by the origin server at the time of each request.

The `Allow` header field MAY be provided with a PUT request to recommend the methods to be supported by the new or modified resource. The server is not required to support these methods and SHOULD include an `Allow` header in the response giving the actual supported methods.

A proxy MUST not modify the `Allow` header field even if it does not understand all the methods specified, since the user agent MAY have other means of communicating with the origin server.

The `Allow` header field does not indicate what methods are implemented at the server level. Servers MAY use the Public response header field (Section 10.32) to describe what methods are implemented on the server as a whole.

10.6 Authorization

A user agent that wishes to authenticate itself with a server--usually, but not necessarily, after receiving a 401 response--MAY do so by including an `Authorization` request-header field with the request. The `Authorization` field value consists of `credentials` containing the authentication information of the user agent for the realm of the resource being requested.

```
Authorization  = "Authorization" ":" credentials
```

HTTP access authentication is described in Section 11. If a request is authenticated and a `realm` specified, the same `credentials` SHOULD be valid for all other requests within this `realm`.

When a shared cache (see section 13.10) receives a request containing an `Authorization` field, it MUST NOT return the corresponding response as a reply to any other request, unless one of the following specific exceptions holds:

1. If the response includes the "`proxy-revalidate`" `Cache-Control` directive, the cache MAY use that response in replying to a subsequent request, but a proxy cache MUST first revalidate it with the origin server, using the request headers from the new request to allow the origin server to authenticate the new request.
2. If the response includes the "`must-revalidate`" `Cache-Control` directive, the cache MAY use that response in replying to a subsequent request, but all caches MUST first revalidate it with the origin server, using the request headers from the new request to allow the origin server to authenticate the new request.
3. If the response includes the "`public`" `Cache-Control` directive, it may be returned in reply to any subsequent request.

10.7 Cache-Control

The `Cache-Control` general-header field is used to specify directives that MUST be obeyed by all caching mechanisms along the request/response chain. The directives specify behavior intended to prevent caches from adversely interfering with the request or response. . These directives typically override the default caching algorithms. Cache directives are unidirectional in that the presence of a directive in a request does not imply that the same directive should be given in the response.

Cache directives must be passed through by a proxy or gateway application, regardless of their significance to that application, since the directives may be applicable to all recipients along the request/response chain. It is not possible to specify a cache-directive for a specific cache.

```
Cache-Control   = "Cache-Control" ":" 1#cache-directive

cache-directive = "public"
                | "private" [ "=" <"> 1#field-name <"> ]
                | "no-cache" [ "=" <"> 1#field-name <"> ]
                | "no-store"
                | "no-transform"
                | "must-revalidate"
                | "proxy-revalidate"
                | "only-if-cached"
                | "max-age" "=" delta-seconds
                | "max-stale" "=" delta-seconds
                | "min-fresh" "=" delta-seconds
                | "min-vers" "=" HTTP-Version

and perhaps
                | "max-uses" "=" 1*DIGIT
                | "use-count" "=" 1*DIGIT
```

When a directive appears without any `1#field-name` parameter, the directive applies to the entire request or response. When such a directive appears with a `1#field-name` parameter, it applies only to the named field or fields, and not to the rest of the request or response. This mechanism supports extensibility; implementations of future versions of the HTTP protocol may apply these directives to header fields not defined in HTTP/1.1.

The cache-control directives can be broken down into these general categories:

- Restrictions on what is cachable; these may only be imposed by the origin server.

- Restrictions on what may be stored by a cache; these may be imposed by either the origin server or the end-user client.
- Modifications of the basic expiration mechanism; these may be imposed by either the origin server or the end-user client.
- Controls over cache revalidation and reload; these may only be imposed by an end-user client.
- Restrictions on the number of times a cache entry may be used, and related demographic reporting mechanisms.
- Miscellaneous restrictions

Caches never add or remove `Cache-Control` directives to requests or responses.

Check: is this true?

10.7.1 SLUSHY: Restrictions on What is Cachable

Unless specifically constrained by a `Cache-Control` directive, a caching system may always store a successful response as a cache entry, may return it without validation if it is fresh, and may return it after successful validation. If there is neither a cache validator nor an explicit expiration time associated with a response, we do not expect it to be cached, but certain caches may violate this expectation (for example, when little or no network connectivity is available) as long as they explicit mark their responses using the `Warning` mechanism describe in section 10.51.

> Note that some HTTP/1.0 caches are known to violate this expectation without providing any `Warning`.

However, in some cases it may be inappropriate for a cache to retain a resource value, or to return it in response to a subsequent request. This may be because absolute semantic transparency is deemed necessary by the service author, or because of security or privacy considerations. Certain `Cache-Control` directives are therefore provided so that the server can indicate that certain resources, or portions thereof, may not be cached regardless of other considerations.

Note that section 10.6 normally prevents a shared cache from saving and returning a response to a previous request if that request included an `Authorization` header.

The following `Cache-Control` response directives add or remove restrictions on what is cachable:

public
: Overrides the restriction in section 10.6 that prevents a shared cache from saving and returning a response to a previous request if that request included an `Authorization` header. However, any other constraints on caching still apply.

private
: Indicates that all or parts of the response message are intended for a single user and MUST NOT be cached by a shared cache. This allows an origin server to state that the specified parts of the response are intended for only one user and are not a valid response for requests by other users. applicable to responses and must not be generated by clients. A private (non-shared) cache may ignore this directive.

  Note: This usage of the word "`private`" only controls where the response may cached, and cannot ensure the privacy of the message content. Note in particular that HTTP/1.0 caches will not recognize or obey this directive.

no-cache
: indicates that all or parts of the response message MUST NOT be cached. This allows an origin server to prevent caching even by caches that have been configured to return stale responses to client requests.

> Note: HTTP/1.0 caches will not recognize or obey this directive.

TBS: precedence relations between public, private, and no-cache.

10.7.2 Restrictions On What May be Stored by a Cache

The "`no-store`" directive applies to the entire message, and may be sent either in a response or in a request. If sent in a request, a cache MUST NOT store any part of either this request or any response to it. If sent in a response, a cache MUST NOT store any part of either this response or the request that elicited it. This directive applies to both non-shared and shared caches.

Even when this directive is associated with a response, users may explicitly store such a response outside of the caching system (e.g., with a "Save As" dialog). History buffers may store such responses as part of their normal operation.

The purpose of this directive is to meet the stated requirements of certain users and service authors who are concerned about accidental releases of information via unanticipated accesses to cache data structures. While the use of this directive may improve privacy in some cases, we caution that it is NOT in any way a reliable or sufficient mechanism for ensuring privacy. In particular, HTTP/1.0 caches will not recognize or obey this directive, malicious or compromised caches may not recognize or obey this directive, and all communications networks may be vulnerable to eavesdropping.

The "`min-vers`" directive applies to the entire message, and may be sent either in a response or in a request. If sent in a request, a cache whose HTTP version number is less than the specified version MUST NOT store any part of either this request or any response to it. If sent in a response, a cache whose HTTP version number is less than the specified version MUST NOT store any part of either this response or the request that elicited it, nor may any cache transmit a stored (non-firsthand) copy of the response to any client with a lower HTTP version number. This directive applies to both non-shared and shared caches, and is made mandatory to allow for future protocol extensions that may affect caching.

> Note that the lowest version that can be sensibly included in a "min-vers" directive is HTTP/1.1, since HTTP/1.0 caches do not obey it.

10.7.3 Modifications of the Basic Expiration Mechanism

The expiration time of a resource may be specified by the origin server using the `Expires` header (see section TBS). Alternatively, it may be specified using the "max-age" directive in a response.

If a response includes both an `Expires` header and a max-age: directive, the max-age: directive overrides the `Expires` header, even if the `Expires` header is more restrictive. This rule allows an origin server to provide, for a given response, a longer expiration time to an HTTP/1.1 (or later) cache than to an HTTP/1.0 cache. This may be useful if certain HTTP/1.0 caches improperly calculate ages or expiration times, perhaps due to badly unsynchronized clocks.

Other directives allow an end-user client to modify the basic expiration mechanism, making it either stricter or looser. These directives may be specified on a request:

max-age Indicates that the client is willing to accept a response whose age is no greater than the specified time in seconds. Unless "`max-stale`" is also included, the client is not willing to accept a stale response. This directive overrides any policy of the cache.

min-fresh 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 that response will still be fresh for at least the specified number of seconds.

max-stale Indicates that the client is willing to accept a response that has exceeded its expiration time by no more than the specified number of seconds. If a cache returns a stale response in response to such a request, it MUST mark it as stale using the `Warning` header.

Note that HTTP/1.0 caches will ignore these directives.

If a cache returns a stale response, either because of a max-stale directive on a request, or because the cache is configured to override the expiration time of a response, the cache MUST attach a `Warning` header to the stale response, using Warning 10 (Response is stale).

10.7.4 SLUSHY: Controls over cache revalidation and reload

Sometimes an end-user client may want or need to insist that a cache revalidate its cache entry with the origin server (and not just with the next cache along the path to the origin server), or to reload its cache entry from the origin server. End-to-end revalidation may be necessary if either the cache or the origin server has overestimated the expiration time of the cached response. End-to-end reload may be necessary if the response value has become corrupted for some reason, and the fact that its validator is up-to-date is irrelevant.

End-to-end revalidation may be requested either when the client does not have its own local cached copy, in which case we call it "unspecified end-to-end revalidation", or when the client does have a local cached copy, in which case we call it "specific end-to-end revalidation."

The client can specify these three kinds of action using `Cache-Control` request directives:

End-to-end reload The request includes "`Cache-Control:` no-cache" or, for compatibility with HTTP/1.0 clients, "Pragma: no-cache". No field names may be included with the "no-cache" directive in a request. The server MUST NOT use a cached copy when responding to such a request.

Specific end-to-end revalidation The request includes "`Cache-Control`: max-age=0", which forces each cache along the path to the origin server to revalidate its own entry, if any, with the next cache or server. The initial request includes a cache-validating conditional with the client's current validator.

Unspecified end-to-end revalidation The request includes "`Cache-Control`: max-age=0", which forces each cache along the path to the origin server to revalidate its own entry, if any, with the next cache or server. The initial request does not include a cache-validating conditional; the first cache along the path (if any) that holds a cache entry for this resource includes a cache-validating conditional with its current validator.

Note that HTTP/1.0 caches will ignore these directives, except perhaps for "`Pragma: no-cache`".

When an intermediate cache is forced, by means of a "`max-age=0`" directive, to revalidate its own cache entry, and the client has supplied its own validator in the request, the supplied validator may differ from the validator currently stored with the cache entry. In this case, the cache may use either validator in making its own request without affecting semantic transparency.

However, the choice of validator may affect performance. The best approach is for the intermediate cache to use its own validator when making its request. If the server replies with 304 (Not Modified), then the cache should return its now validated copy to the client with a 200 (OK) response. If the server replies with a new Entity-body and cache validator, however, the intermediate cache should compare the returned validator with the one provided in the client's request, using the strong comparison function. If the client's validator is equal to the origin server's, then the intermediate cache simply returns 304 (Not Modified). Otherwise, it returns the new Entity-body with a 200 (OK) response.

If a request includes the "`no-cache`" directive, it should not include "`fresh-min`", "`max-stale`", or "`max-age`".

In some cases, such as times of extremely poor network connectivity, a client may want a cache to return only those responses that it currently has stored, and not to reload or revalidate with the origin server. To do this, the client may include the "`only-if-cached`" directive in a request. If it receives this directive, a cache SHOULD either respond using a cached value that is consistent with the other constraints of the request, or respond with a 504 (Gateway Timeout) status. However, if a group of caches is being operated as a unified system with good internal connectivity, such a request MAY be forwarded within that group of caches.

Because a cache may be configured to ignore a server's specified expiration time, and because a client request may include a max-stale directive, which has a similar effect, the protocol also includes a mechanism for the origin server to require revalidation of a cache entry on any subsequent use. When the “must-revalidate” directive is present in a response received by a cache, that cache MUST NOT use the value after it becomes stale to respond to a subsequent request without first revalidating it with the origin server. (I.e., the cache must do an end-to-end revalidation every time.)

The “must-revalidate” directive is necessary to support reliable operation for cookies and certain other protocol features. In all circumstances an HTTP/1.1 cache MUST obey the “must-revalidate” directive; in particular, if the cache cannot reach the origin server for any reason, it MUST generate a 504 (Gateway Timeout) response. Note that HTTP/1.0 caches will ignore this directive.

Servers should send the “must-revalidate” directive if and only if failure to revalidate a request on the entity could result in significantly incorrect operation, such as a silently unexecuted financial transaction. Recipients MUST NOT take any automated action that violates this directive, and MUST NOT automatically provide an unvalidated copy of the entity if revalidation fails.

Although this is not recommended, user agents operating under severe connectivity constraints may violate this directive but if so, MUST explicitly warn the user that an unvalidated response has been provided. The warning MUST be provided on each unvalidated access, and SHOULD require explicit user confirmation.

The “proxy-revalidate” directive has the same meaning as the “must-revalidate” directive, except that it does not apply to user-agent caches. This directive is meant to support digest authentication.

10.7.5 FLUID: Restrictions on use count and demographic reporting

This section is highly debatable and is likely to be removed to a separate I.D.

The “max-uses” response directive allows a cache to use a response at most a certain limited number of times. For example, “max-uses=10” means that the response should be returned in reply to the current request, and may be returned in reply to no more than nine subsequent requests (subject to other caching constraints), unless revalidated.

A cache may subdivide its remaining use-count among several of its own clients. For example, if the incoming response includes “max-uses=10”, the recipient may forward this as two responses, each with “max-uses=5”. The idea is that the total number of uses allowed in a cache hierarchy should not exceed the specified limit. (The heuristics a cache uses to sub-allocate its max-uses value are beyond the scope of the HTTP spec.)

The “use-count” request directive allows a cache to tell a server how many times it has actually used the cache entry specified in the associated request. If a cache receives a use-count value from one of its clients, and it has a corresponding cache entry, it should add the incoming use-count to its local count.

When a cache removes an entry, it MAY first send a HEAD request on the associated URI, including its use-count value, to inform the server of the actual use-count. If the server has sent a max-uses limit, the cache SHOULD perform this notification.

A cache that is willing to perform such notifications and that is willing to obey the max-uses limit SHOULD send a ``use-count=0" directive on its first (non-conditional) request on a resource. This informs the server that the cache intends to use these two directives in the manner described here.

10.7.6 Miscellaneous restrictions

In certain circumstances, an intermediate cache (proxy) may find it useful to convert the encoding of an entity body. For example, a proxy might use a compressed content-coding to transfer the body to a client on a slow link.

Because end-to-end authentication of entity bodies and/or entity headers relies on the specific encoding of these values, such transformations may cause authentication failures. Therefore, an intermediate cache MUST NOT change the encoding of an entity body if the response includes the “no-transform” directive.

10.8 Connection

HTTP version 1.1 provides a new request and response header field called “`Connection`”. This header field allows the client and server to specify options which should only exist over that particular connection and MUST NOT be communicated by proxies over further connections. The connection header field MAY have multiple tokens separated by commas (referred to as connection-tokens).

HTTP version 1.1 proxies MUST parse the `Connection` header field and for every connection-token in this field, remove a corresponding header field from the request before the request is forwarded. The use of a connection option is specified by the presence of a connection token in the `Connection` header field, not by the corresponding additional header field (which may not be present).

When a client wishes to establish a persistent connection it MUST send a “`Persist`” connection-token:

```
Connection: persist
```

The Connection header has the following grammar:

```
Connection-header = "Connection" ":"
                          connection-token 0#( "," connection-token )
```

10.8.1 Persist

When the `Persist` connection-token has been transmitted with a request or a response a `Persist` header field MAY also be included. The `Persist` header field takes the following form:

```
Persist-header = "Persist" ":" 0#pers-param

pers-param = param-name "=" value
```

The `Persist` header itself is optional, and is used only if a parameter is being sent. HTTP/1.1 does not define any parameters.

If the `Persist` header is sent, the corresponding connection token MUST be transmitted. The `Persist` header MUST be ignored if received without the connection token.

10.9 Content-Base

The `Content-Base` entity-header field may be used to specify the base URI for resolving relative URLs within the entity. This header field is described as "Base" in RFC 1808 [11], which is expected to be revised soon.

```
Content-Base           = "Content-Base" ":" absoluteURI
```

If no `Content-Base` field is present, the base URI of an entity is defined either by its `Content-Location` or the URI used to initiate the request, in that order of precedence. Note, however, that the base URI of the contents within the entity body may be redefined within that entity body.

10.10 Content-Encoding

The `Content-Encoding` entity-header field is used as a modifier to the `media-type`. When present, its value indicates what additional content codings have been applied to the resource, and thus what decoding mechanisms MUST be applied in order to obtain the `media-type` referenced by the `Content-Type` header field. `Content-Encoding` is primarily used to allow a document to be compressed without losing the identity of its underlying media type.

```
Content-Encoding        = "Content-Encoding" ":" 1#content-coding
```

Content codings are defined in Section 3.5. An example of its use is

```
Content-Encoding: gzip
```

The `Content-Encoding` is a characteristic of the resource identified by the `Request-URI`. Typically, the resource is stored with this encoding and is only decoded before rendering or analogous usage.

If multiple encodings have been applied to a resource, the content codings MUST be listed in the order in which they were applied. Additional information about the encoding parameters MAY be provided by other Entity-Header fields not defined by this specification.

10.11 Content-Language

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

```
Content-Language        = "Content-Language" ":" 1#language-tag
```

Language tags are defined in Section 3.10. The primary purpose of `Content-Language` is to allow a selective consumer to identify and differentiate resources according to the consumer's own preferred language. Thus, if the body content is intended only for a Danish-literate audience, the appropriate field is

```
Content-Language: dk
```

If no `Content-Language` is specified, the default is that the content is intended for all language audiences. This may 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 an entity 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` should only include "en".

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

10.12 Content-Length

The `Content-Length` entity-header field indicates the size of the `Entity-Body`, in decimal number of octets, sent to the recipient or, in the case of the `HEAD` method, the size of the `Entity-Body` that would have been sent had the request been a `GET`.

```
Content-Length = "Content-Length" ":" 1*DIGIT
```

An example is

```
Content-Length: 3495
```

Applications SHOULD use this field to indicate the size of the `Entity-Body` to be transferred, regardless of the media type of the entity. A valid `Content-Length` field value is required on all HTTP/1.1 request messages containing an entity body.

Any `Content-Length` greater than or equal to zero is a valid value. Section 7.2.2 describes how to determine the length of an `Entity-Body` if a `Content-Length` is not given.

> Note: The meaning of this field is significantly different from the corresponding definition in MIME, where it is an optional field used within the “message/external-body” content-type. In HTTP, it SHOULD be used whenever the entity's length can be determined prior to being transferred.

10.13 Content-MD5

The `Content-MD5` entity-header field is an MD5 digest of the entity-body, as defined in RFC 1864 [23], for the purpose of providing an end-to-end message integrity check (MIC) of the entity-body. (Note: an MIC is good for detecting accidental modification of the entity-body in transit, but is not proof against malicious attacks.)

```
ContentMD5      = "Content-MD5" ":" md5-digest
md5-digest      = <base64 of 128 bit MD5 digest as per RFC 1864>
```

The `Content-MD5` header may be generated by an origin server to function as an integrity check of the entity-body. Only origin-servers may generate the `Content-MD5` header field; proxies and gateways MUST NOT generate it, as this would defeat its value as an end-to-end integrity check. Any recipient of the entity-body, including gateways and proxies, MAY check that the digest value in this header field matches that of the entity-body as received.

The MD5 digest is computed based on the content of the entity body, including any `Content-Encoding` that has been applied, but not including any `Transfer-Encoding`. If the entity is received with a `Transfer-Encoding`, that encoding must be removed prior to checking the `Content-MD5` value against the received entity.

This has the result that the digest is computed on the octets of the entity body exactly as, and in the order that, they would be sent if no `Transfer-Encoding` were being applied.

HTTP extends RFC 1864 to permit the digest to be computed for MIME composite media-types (e.g., multipart/* and message/rfc822), but this does not change how the digest is computed as defined in the preceding paragraph.

> Note: There are several consequences of this. The entity-body for composite types many contain many body-parts, each with its own MIME and HTTP headers (including

`Content-MD5`, `Content-Transfer-Encoding`, and `Content-Encoding` headers). If a body-part has a `Content-Transfer-Encoding` or `Content-Encoding` header, it is assumed that the content of the body-part has had the encoding applied, and the body-part is included in the `Content-MD5` digest as is -- i.e., after the application. Also, the HTTP `Transfer-Encoding` header makes no sense within body-parts; if it is present, it is ignored -- i.e. treated as ordinary text.

Note: while the definition of `Content-MD5` is exactly the same for HTTP as in RFC 1864 for MIME entity-bodies, there are several ways in which the application of `Content-MD5` to HTTP entity-bodies differs from its application to MIME entity-bodies. One is that HTTP, unlike MIME, does not use `Content-Transfer-Encoding`, and does use `Transfer-Encoding` and `Content-Encoding`. Another is that HTTP more frequently uses binary content types than MIME, so it is worth noting that in such cases, the byte order used to compute the digest is the transmission byte order defined for the type. Lastly, HTTP allows transmission of text types with any of several line break conventions and not just the canonical form using CR-LF. Conversion of all line breaks to CR-LF should not be done before computing or checking the digest: the line break convention used in the text actually transmitted should be left unaltered when computing the digest.

10.14 SLUSHY Content-Range

The Content-Range header is sent with a partial entity body to specify where in the full entity body the partial body should be inserted. It also indicates the total size of the entity.

```
Content-Range = "Content-Range" ":" content-range-spec
```

When an HTTP message includes the content of a single range (for example, a response to a request for a single range, or to request for a set of ranges that overlap without any holes), this content is transmitted with a Content-Range header, and a Content-length header showing the number of bytes actually transferred.

For example,

```
HTTP/1.0 206 Partial content

Date: Wed, 15 Nov 1995 06:25:24 GMT

Last-modified: Wed, 15 Nov 1995 04:58:08 GMT

Content-range: 21010-47021/47022

Content-length: 26012

Content-type: image/gif
```

10.14.1 MIME multipart/byteranges content-type

When an HTTP message includes the content of multiple ranges (for example, a response to a request for multiple non-overlapping ranges), these are transmitted as a multipart MIME message. The multipart MIME content-type used for this purpose is defined in this specification to be "multipart/byteranges".

The MIME multipart/byteranges content-type includes two or more parts, each with its own Content-type and Content-Range fields. The parts are separated using a MIME boundary parameter.

For example:

```
 HTTP/1.0 206 Partial content
 Date: Wed, 15 Nov 1995 06:25:24 GMT
 Last-modified: Wed, 15 Nov 1995 04:58:08 GMT
 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—
```

10.14.2 Additional rules for Content-Range

A client that cannot decode a MIME multipart/byteranges message should not ask for multiple byte-ranges in a single request.

When a client requests multiple byte-ranges in one request, the server SHOULD return them in the order that they appeared in the request.

If the server ignores a byte-range-spec because it is invalid, or because it specifies a range that starts beyond the end of the entity, it may omit the corresponding Content-Range field and partial entity body.

If none of the byte-range-spec values in a request specify part of the current entity (i.e., start before the last byte of the entity), then the server should return a status of 207 (Range Out Of Bounds).

10.15 Content-Type

The `Content-Type` entity-header field indicates the media type of the `Entity-Body` sent to the recipient or, in the case of the `HEAD` method, the media type that would have been sent had the request been a `GET`.

```
Content-Type   = "Content-Type" ":" media-type
```

Media types are defined in Section 3.7. An example of the field is

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

Further discussion of methods for identifying the media type of an entity is provided in Section 7.2.1.

10.16 Content-Location

The `Content-Location` entity-header field is used to define the location of the specific resource associated with the entity enclosed in the message. A server SHOULD provide a `Content-Location` if, when including an entity in response to a GET request on a negotiated resource, the entity corresponds to a specific, non-negotiated location which can be accessed via the `Content-Location` URI. A server SHOULD provide a `Content-Location` with any 200 (OK) response which was internally (not visible to the client) redirected to a resource other than the one identified by the request and for which correct interpretation of that resource MAY require knowledge of its actual location. The recipient MAY make future requests on this location instead of on the `Request-URI`.

```
Content-Location = "Content-Location" ":" absoluteURI
```

If no `Content-Base` header field is present, the value of `Content-Location` also defines the base URL for the entity (see Section 10.9).

> Note: Since the `Content-Location` field re-interprets the source of an entity, recipients must take care in not allowing a "spoofed" location to deny access to the real resource. This is described in Section 15.9.

10.17 Date

The `Date` general-header field represents the date and time at which the message was originated, having the same semantics as `orig-date` in RFC 822. The field value is an `HTTP-date`, as described in Section 3.3.

```
Date           = "Date" ":" HTTP-date
```

An example is

```
Date: Tue, 15 Nov 1994 08:12:31 GMT
```

If a message is received via direct connection with the user agent (in the case of requests) or the origin server (in the case of responses), then the date can be assumed to be the current date at the receiving end. However, since the date--as it is believed by the origin--is important for evaluating cached responses, origin servers SHOULD always include a `Date` header. Clients SHOULD only send a `Date` header field in messages that include an entity body, as in the case of the PUT and POST requests, and even then it is optional. A received message which does not have a `Date` header field SHOULD be assigned one by the recipient if the message will be cached by that recipient or gatewayed via a protocol which requires a `Date`.

In theory, the date SHOULD represent the moment just before the entity is generated. In practice, the date can be generated at any time during the message origination without affecting its semantic value.

> Note: An earlier version of this document incorrectly specified that this field SHOULD contain the creation date of the enclosed `Entity-Body`. This has been changed to reflect actual (and proper) usage.

Origin servers MUST send a `Date` field in every response. However, if a cache receives a response without a `Date` field, it SHOULD attach one with the cache's best estimate of the time at which the response was originally generated.

The format of the `Date` is an absolute date and time as defined by HTTP-date in Section 3.3; it MUST be in RFC1123-date format.

10.19 SLUSHY Expires

The `Expires` entity-header field gives the date/time after which the entity should be considered stale. A stale cache entry may not normally be returned by a cache (either a proxy cache or an end-user cache) unless it is first validated with the origin server (or with an intermediate cache that has a fresh copy of the resource). See section 13.2 for further discussion of the expiration 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 format is an absolute date and time as defined by HTTP-date in Section 3.3; it MUST be in rfc1123-date format:

```
Expires = "Expires" ":" HTTP-date
```

An example of its use is

```
Expires: Thu, 01 Dec 1994 16:00:00 GMT
```

> Note: if a response includes a `Cache-Control` field with the max-age directive, that directive overrides the `Expires` field.

HTTP/1.1 clients and caches MUST treat other invalid date formats, especially including the value “0”, as in the past (i.e., “already expired”).

To mark a response as “already expired,” an origin server should use an `Expires` date that is equal to the `Date` header value. (See the rules for expiration calculations in section 13.2.7.)

To mark a response as “never expires,” an origin server should use `Expires` date approximately one year from the time the response is generated. HTTP/1.1 servers should not send `Expires` dates more than one year in the future.

10.20 Via

The `Via` general-header field MUST be used by gateways and proxies to indicate the intermediate protocols and recipients between the user agent and the server on requests, and between the origin server and the client on responses. It is analogous to the “Received” field of RFC 822 [9]and is intended to be used for tracking message forwards, avoiding request loops, and identifying the protocol capabilities of all senders along the request/response chain.

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

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

received-by       = ( host [ ":" port ] ) | pseudonym
pseudonym         = token
```

The received-protocol indicates the protocol version of the message received by the server or client along each segment of the request/response chain. The received-protocol version is appended to the `Via` field value when the message is forwarded so that information about the protocol capabilities of upstream applications remains visible to all recipients.

The protocol-name is optional if and only if it would be "HTTP". The received-by field 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, it MAY be replaced by a pseudonym. If the port is not given, it MAY be assumed to be the default port of the received-protocol.

Multiple `Via` field values represent each proxy or gateway that has forwarded the message. Each recipient MUST append their information such that the end result is ordered according to the sequence of forwarding applications.

Comments MAY be used in the `Via` header field to identify the software of the recipient proxy or gateway, analogous to the User-Agent and Server header fields. However, all comments in the `Via` field are optional and MAY be removed by any recipient 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 nowhere.com, which completes the request by forwarding it to the origin server at www.ics.uci.edu. The request received by www.ics.uci.edu would then have the following `Via` header field:

```
Via: 1.0 fred, 1.1 nowhere.com (Apache/1.1)
```

Proxies and gateways used as a portal through a network firewall SHOULD NOT, by default, forward the names and ports of hosts within the firewall region. This information SHOULD only be propagated if explicitly enabled. If not enabled, the received-by host of any host behind the firewall SHOULD be replaced by an appropriate pseudonym for that host.

For organizations that have strong privacy requirements for hiding internal structures, a proxy MAY combine an ordered subsequence of `Via` header field entries with identical received-protocol values into a single such entry. 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
```

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

> **Note:** The `Via` header field replaces the `Forwarded` header field which was present in earlier drafts of this specification.

10.21 From

The `From` request-header field, if given, SHOULD contain an Internet e-mail address for the human user who controls the requesting user agent. The address SHOULD be machine-usable, as defined by `mailbox` in RFC 822 [9] (as updated by RFC 1123 [8]):

```
From          = "From" ":" mailbox
```

An example is:

```
From: webmaster@w3.org
```

This header field MAY be used for logging purposes and as a means for identifying the source of invalid or unwanted requests. It SHOULD NOT be used as an insecure form of access protection. The interpretation of this field is that the request is being performed on behalf of the person given, who accepts responsibility for the `method` performed. In particular, robot agents SHOULD include this header so that the person responsible for running the robot can be contacted if problems occur on the receiving end.

The Internet e-mail address in this field MAY be separate from the Internet host which issued the request. For example, when a request is passed through a proxy the original issuer's address SHOULD be used.

> Note: The client SHOULD not send the `From` header field without the user's approval, as it may conflict with the user's privacy interests or their site's security policy. It is strongly recommended that the user be able to disable, enable, and modify the value of this field at any time prior to a request.

10.22 Host

The `Host` request-header field specifies the Internet host and port number of the resource being requested, as obtained from the original URL given by the user or referring resource (generally an HTTP URL, as described in Section 3.2.2). The `Host` field value MUST represent the network location of the origin server or gateway given by the original URL. This allows the origin server or gateway to differentiate between internally-ambiguous URLs, such as the root "/" URL of a server for multiple host names on a single IP address.

```
Host = "Host" ":" host [ ":" port ]     ; see Section 3.2.2
```

A "host" without any trailing port information implies the default port for the service requested (e.g., "80" for an HTTP URL). For example, a request on the origin server for `<http://www.w3.org/pub/WWW/>` MUST include:

```
GET /pub/WWW/ HTTP/1.1

Host: www.w3.org
```

The `Host` header field MUST be included in all HTTP/1.1 request messages on the Internet (i.e., on any message corresponding to a request for a URL which includes an Internet host address for the service being requested). If the `Host` field is not already present, an HTTP/1.1 proxy MUST add a `Host` field to the request message prior to forwarding it on the Internet. All Internet-based HTTP/1.1 servers MUST respond with a 400 status code to any HTTP/1.1 request message which lacks a `Host` header field.

10.23 If-Modified-Since

The `If-Modified-Since` request-header field is used with the `GET` method to make it conditional: if the requested resource has not been modified since the time specified in this field, a copy of the resource will not be returned from the server; instead, a 304 (not modified) response will be returned without any `Entity-Body`.

```
If-Modified-Since = "If-Modified-Since" ":" HTTP-date
```

An example of the field is:

```
If-Modified-Since: Sat, 29 Oct 1994 19:43:31 GMT
```

A GET method with an `If-Modified-Since` header and no `Range` header requests that the identified resource be transferred only if it has been modified since the date given by the `If-Modified-Since` header. The algorithm for determining this includes the following cases:

a)
: If the request would normally result in anything other than a 200 (ok) status, or if the passed `If-Modified-Since` date is invalid, the response is exactly the same as for a normal `GET`. A date which is later than the server's current time is invalid.

b)
: If the resource has been modified since the `If-Modified-Since` date, the response is exactly the same as for a normal `GET`.

c)
: If the resource has not been modified since a valid `If-Modified-Since` date, the server MUST return a 304 (not modified) response.

The purpose of this feature is to allow efficient updates of cached information with a minimum amount of transaction overhead.

> Note that the `Range` request-header field modifies the meaning of `If-Modified-Since`; see section 13.9 for full details.

> Note that `If-Modified-Since` is ignored for varying resources.

> Note that `If-Modified-Since` times are interpreted by the server, whose clock may not be synchronized with the client.

> Note that if a client uses an arbitrary date in the `If-Modified-Since` header instead of a date taken from the `Last-Modified` header for the same request, the client should be aware of the fact that this date is interpreted in the server's understanding of time. The client should consider unsynchronized clocks and rounding problems due to the different representations of time between the client and server. This includes the possibility of race conditions if the document has changed between the time it was first request and the `If-Modified-Since` date of a subsequent request, and the possibility of clock-skew-related problems if the `If-Modified-Date` date is derived from the client's clock without correction to the server's clock. Corrections for different time bases between client and server are at best approximate due to network latency.

10.25 Last-Modified

The `Last-Modified` entity-header field indicates the date and time at which the sender believes the resource was last modified. The exact semantics of this field are defined in terms of how the recipient SHOULD interpret it: if the recipient has a copy of this resource which is older than the date given by the `Last-Modified` field, that copy SHOULD be considered stale.

```
Last-Modified  = "Last-Modified" ":" HTTP-date
```

An example of its use is

```
Last-Modified: Tue, 15 Nov 1994 12:45:26 GMT
```

The exact meaning of this header field depends on the implementation of the sender and the nature of the original resource. For files, it may be just the file system last-modified time. For entities with dynamically included parts, it may be the most recent of the set of last-modify times for its component parts. For database gateways, it may be the last-update time stamp of the record. For virtual objects, it may be the last time the internal state changed.

An origin server MUST not send a `Last-Modified` date which is later than the server's time of message origination. In such cases, where the resource's last modification would indicate some time in the future, the server MUST replace that date with the message origination date.

An origin server should obtain the `Last-Modified` value of the entity as close as possible to the time that it generates the `Date` value of its response. This allows a recipient to make an accurate assessment of the entity's modification time, especially if the entity changes near the time that the response is generated.

10.27 Location

The `Location` response-header field is used to redirect the recipient to a location other than the `Request-URI` for completion of the request or identification of a new resource. For 201 responses, the `Location` is that of the new resource which was created by the request. For 3xx responses, the location SHOULD indicate the server's preferred URL for automatic redirection to the resource. The field value consists of a single absolute URL.

```
Location       = "Location" ":" absoluteURI
```

An example is

```
Location: http://www.w3.org/pub/WWW/People.html
```

> Note: The `Content-Location` header field (Section 10.16) differs from `Location` in that the former identifies the original location of the entity enclosed in the request. It is therefore possible for a response to contain header fields for both `Location` and `Content-Location`.

10.29 Pragma

The Pragma general-header field is used to include implementation-specific directives that may apply to any recipient along the request/response chain. All pragma directives specify optional behavior from the viewpoint of the protocol; however, some systems MAY require that behavior be consistent with the directives.

```
Pragma                 = "Pragma" ":" 1#pragma-directive

pragma-directive       = "no-cache" | extension-pragma
extension-pragma       = token [ "=" word ]
```

When the "`no-cache`" directive is present in a request message, an application SHOULD forward the request toward the origin server even if it has a cached copy of what is being requested. This pragma directive has the same semantics as the "no-cache" cache-directive (see Section 10.8) and is defined here for backwards compatibility with HTTP/1.0. Clients SHOULD include both header fields when a "no-cache" request is sent to a server not known to be HTTP/1.1 compliant.

Pragma directives MUST be passed through by a proxy or gateway application, regardless of their significance to that application, since the directives may be applicable to all recipients along the request/response chain. It is not possible to specify a pragma for a specific recipient; however, any pragma directive not relevant to a recipient SHOULD be ignored by that recipient.

HTTP/1.1 clients SHOULD NOT send the `Pragma` request header. HTTP/1.1 caches SHOULD treat "`Pragma: no-cache`" as if the client had sent "`Cache-control: no-cache`". No new `Pragma` directives will be defined in HTTP.

10.30 Proxy-Authenticate

The `Proxy-Authenticate` response-header field MUST be included as part of a 407 (proxy authentication required) response. The field value consists of a challenge that indicates the authentication scheme and parameters applicable to the proxy for this `Request-URI`.

```
Proxy-Authentication    = "Proxy-Authentication" ":" challenge
```

The HTTP access authentication process is described in Section 11. Unlike WWW-Authenticate, the `Proxy-Authenticate` header field applies only to the current connection and MUST not be passed on to downstream clients.

10.31 Proxy-Authorization

The `Proxy-Authorization` request-header field allows the client to identify itself (or its user) to a proxy which requires authentication. The `Proxy-Authorization` field value consists of credentials containing the authentication information of the user agent for the proxy and/or realm of the resource being requested.

```
Proxy-Authorization     = "Proxy-Authorization" ":" credentials
```

The HTTP access authentication process is described in Section 11. Unlike Authorization, the `Proxy-Authorization` applies only to the current connection and MUST not be passed on to upstream servers. If a request is authenticated and a realm specified, the same credentials SHOULD be valid for all other requests within this realm.

10.32 Public

The `Public` response-header field lists the set of non-standard methods supported by the server. The purpose of this field is strictly to inform the recipient of the capabilities of the server regarding unusual methods. The methods listed may or may not be applicable to the `Request-URI`; the Allow header field (Section 10.5) SHOULD be used to indicate methods allowed for a particular URI. This does not prevent a client from trying other methods. The field value SHOULD not include the methods predefined for HTTP/1.1 in Section 5.1.1.

```
Public         = "Public" ":" 1#method
```

Example of use:

```
Public: OPTIONS, MGET, MHEAD
```

This header field applies only to the server directly connected to the client (i.e., the nearest neighbor in a chain of connections). If the response passes through a proxy, the proxy MUST either remove the `Public` header field or replace it with one applicable to its own capabilities.

10.33 Range

HTTP retrieval requests using conditional or unconditional GET methods may request one or more sub-ranges of the entity, instead of the entire entity. This is done using the Range request header:

```
Range = "Range" ":" ranges-specifier
```

A server MAY ignore the Range header. However, HTTP/1.1 origin servers and intermediate caches SHOULD support byte ranges whenever possible, since this supports efficient recovery from partially failed transfers, and it supports efficient partial retrieval of large entities.

I the server supports the `Range` header and the specified range or ranges are appropriate for the entity:

- The presence of a `Range` header in an unconditional GET modifies what is returned if the GET is otherwise successful. In other words, the response carries a status code of 206 (Partial content) instead of 200 (OK).
- The presence of a `Range` header in a conditional GET (a request using one or both of `If-Modified-Since` and `If-Invalid`, or one or both of `If-Unmodified-Since` and `If-Valid`) modifies what is returned if the GET is otherwise successful and the condition is true. It does not affect the 304 (Not Modified) response returned if the conditional is false.

In some cases, it may be more appropriate to use the `Range-If` header (see section 10.104) instead of the `Range` header.

10.34 Referer

The `Referer`(sic) request-header field allows the client to specify, for the server's benefit, the address (URI) of the resource from which the `Request-URI` was obtained. This allows a server to generate lists of back-links to resources for interest, logging, optimized caching, etc. It also allows obsolete or mistyped links to be traced for maintenance. The `Referer` field MUST not be sent if the `Request-URI` was obtained from a source that does not have its own URI, such as input from the user keyboard.

```
Referer        = "Referer" ":" ( absoluteURI | relativeURI )
```

Example:

```
Referer: http://www.w3.org/hypertext/DataSources/Overview.html
```

If a partial URI is given, it SHOULD be interpreted relative to the `Request-URI`. The URI MUST not include a fragment.

> Note: Because the source of a link may be private information or may reveal an otherwise private information source, it is strongly recommended that the user be able to select whether or not the `Referer` field is sent. For example, a browser client could have a toggle switch for browsing openly/anonymously, which would respectively enable/disable the sending of `Referer` and `From` information.

10.36 Retry-After

The `Retry-After` response-header field can be used with a 503 (service unavailable) response to indicate how long the service is expected to be unavailable to the requesting client. The value of this field can be either an HTTP-date or an integer number of seconds (in decimal) after the time of the response.

```
Retry-After    = "Retry-After" ":" ( HTTP-date | delta-seconds )
```

Two examples of its use are

```
Retry-After: Wed, 14 Dec 1994 18:22:54 GMT
Retry-After: 120
```

In the latter example, the delay is 2 minutes.

10.37 Server

The `Server` response-header field contains information about the software used by the origin server to handle the request. The field can contain multiple product tokens (Section 3.8) and comments identifying the server and any significant subproducts. By convention, the product tokens are listed in order of their significance for identifying the application.

```
Server         = "Server" ":" 1*( product | comment )
```

Example:

```
Server: CERN/3.0 libwww/2.17
```

If the response is being forwarded through a proxy, the proxy application MUST not add its data to the product list. Instead, it SHOULD include a `Via` field (as described in Section 10.20).

> Note: Revealing the specific software version of the server may allow the server machine to become more vulnerable to attacks against software that is known to contain security holes. Server implementers are encouraged to make this field a configurable option.

10.38 Title

The `Title` entity-header field indicates the title of the entity

```
Title          = "Title" ":" *TEXT
```

An example of the field is

```
Title: Hypertext Transfer Protocol -- HTTP/1.1
```

This field is isomorphic with the <TITLE> element in HTML [5].

10.39 Transfer Encoding

The `Transfer-Encoding` general-header field indicates what (if any) type of transformation has been applied to the message body in order to safely transfer it between the sender and the recipient. This differs from the `Content-Encoding` in that the transfer coding is a property of the message, not of the original resource.

```
Transfer-Encoding      = "Transfer-Encoding" ":" 1#transfer-coding
```

Transfer codings are defined in Section 3.6. An example is:

```
Transfer-Encoding: chunked
```

Many older HTTP/1.0 applications do not understand the `Transfer-Encoding` header.

10.41 Upgrade

The `Upgrade` general-header allows the client to specify what additional communication protocols it supports and would like to use if the server finds it appropriate to switch protocols. The server MUST use the `Upgrade` header field within a 101 (switching protocols) response to indicate which protocol(s) are being switched.

```
Upgrade         = "Upgrade" ":" 1#product
```

For example,

```
Upgrade: HTTP/2.0, SHTTP/1.3, IRC/6.9, RTA/x11
```

The `Upgrade` header field is intended to provide a simple mechanism for transition from HTTP/1.1 to some other, incompatible protocol. It does so by allowing the client to advertise its desire to use another protocol, such as a later version of HTTP with a higher major version number, even though the current request has been made using HTTP/1.1. This eases the difficult transition between incompatible protocols by allowing the client to initiate a request in the more commonly supported protocol while indicating to the server that it would like to use a “better” protocol if available (where “better” is determined by the server, possibly according to the nature of the method and/or resource being requested).

The `Upgrade` header field only applies to switching application-layer protocols upon the existing transport-layer connection. `Upgrade` cannot be used to insist on a protocol change; its acceptance and use by the server is optional. The capabilities and nature of the application-layer communication after the protocol change is entirely dependent upon the new protocol chosen, although the first action after changing the protocol MUST be a response to the initial HTTP request containing the `Upgrade` header field.

The `Upgrade` header field only applies to the immediate connection. Therefore, the “upgrade” keyword MUST be supplied within a `Connection` header field (Section 10.8) whenever `Upgrade` is present in an HTTP/1.1 message.

The `Upgrade` header field cannot be used to indicate a switch to a protocol on a different connection. For that purpose, it is more appropriate to use a 301, 302, 303, or 305 redirection response.

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 3.1 and future updates to this specification. Any token can be used as a protocol name; however, it will only be useful if both the client and server associate the name with the same protocol.

10.43 User-Agent

The `User-Agent` request-header field contains information about the user agent originating the request. This is for statistical purposes, the tracing of protocol violations, and automated recognition of user agents for the sake of tailoring responses to avoid particular user agent limitations. Although it is not required, user agents SHOULD include this field with requests. The field can contain multiple product tokens

(Section 3.8) and comments identifying the agent and any subproducts which form a significant part of the user agent. By convention, the product tokens are listed in order of their significance for identifying the application.

```
User-Agent     = "User-Agent" ":" 1*( product | comment )
```

Example:

```
User-Agent: CERN-LineMode/2.15 libwww/2.17b3
```

10.44 WWW-Authenticate

The `WWW-Authenticate` response-header field MUST be included in 401 (unauthorized) response messages. The field value consists of at least one `challenge` that indicates the authentication scheme(s) and parameters applicable to the `Request-URI`.

```
WWW-Authenticate        = "WWW-Authenticate" ":" 1#challenge
```

The HTTP access authentication process is described in Section 11. User agents MUST take special care in parsing the `WWW-Authenticate` field value if it contains more than one challenge, or if more than one `WWW-Authenticate` header field is provided, since the contents of a challenge may itself contain a comma-separated list of authentication parameters.

10.45 Max-Forwards

[JG17]The `Max-Forwards` general-header field may be used with the TRACE method (Section 8.12) to limit the number of times that a proxy or gateway can forward the request to the next inbound server. This can be useful when the client is attempting to trace a request chain which appears to be failing or looping in mid-chain.

```
Max-Forwards   = "Max-Forwards" ":" 1*DIGIT
```

The `Max-Forwards` value is a decimal integer indicating the remaining number of times this request message may be forwarded.

Each proxy or gateway recipient of a TRACE request containing a `Max-Forwards` header field SHOULD check and update its value prior to forwarding the request. If the received value is zero (0), the recipient SHOULD NOT forward the request; instead, it SHOULD respond as the final recipient with a 200 response containing the received request message as the response entity body (as described in Section 8.12). If the received `Max-Forwards` value is greater than zero, then the forwarded message SHOULD contain an updated `Max-Forwards` field with a value decremented by one (1).

The `Max-Forwards` header field SHOULD be ignored for all other methods defined by this specification and for any extension methods for which it is not explicitly referred to as part of that method definition.

10.46 Age

Caches transmit age values using:

```
Age = "Age" ":" age-value
age-value = delta-seconds
```

Age values are non-negative decimal integers, representing time in seconds.

If a cache receives a value larger than the largest positive integer it can represent, or if any of its age calculations overflows, it MUST NOT transmit an `Age` header. Otherwise, HTTP/1.1 caches MUST send an `Age` header in every response. Caches SHOULD use a representation with at least 31 bits of range.

10.47 CVal

The `CVal` header is used to transmit opaque cache validators in HTTP/1.1 responses.

```
CVal = "CVal" ":" cval-info
cval-info = opaque-validator [ ";" variant-id ]
```

Examples:

```
CVal: "xyzzy"
CVal: "xyzzy"/W
CVal: "xyzzy";3
CVal: "xyzzy"/W;3
CVal: ""
```

Note that the variant-id is not part of the opaque validator. The `CVal` field is used to transmit a variant-id simply as a matter of compact representation of responses.

TBS: does the protocol allow the combination of a null validator and a variant-ID?

10.48 If-Invalid

The `If-Invalid` request-header field is used with a method to make it conditional. A client that has a cache entry for the relevant entity supplies the associated validator using the `If-Invalid` header; if this validator matches the server's current validator for the entity, the server SHOULD return a 304 (Not modified) response without any `Entity-Body`.

If the validators do not match, the server should treat the request as if the `If-Invalid` header was not present.

See section 13.3.3 for rules on how to determine if two validators match.

If the `If-Invalid` header is used to make a conditional request on varying resource, it may be used to pass a set of validators. This is done using the variant-set mechanism if the client has variant IDs for the corresponding cache entries (see sections 13.8.3 and 3.16), or the validator-set mechanism if the client has no variant IDs (see sections 13.8.4 and 3.15).

```
If-Invalid = "If-Invalid" ":" if-invalid-rhs
if-invalid-rhs = variant-set | validator-set
```

Examples of single-entity form:

```
 If-Invalid: "xyzzy"
 If-Invalid: "xyzzy"/W
```

Examples of multiple-entity form:

```
If-Invalid: "xyzzy";4
If-Invalid: "xyzzy";3, "r2d2xxxx";5, "c3piozzzz";7
If-Invalid: "xyzzy"/W;3, "r2d2xxxx"/W;5, "c3piozzzz"/W;7
If-Invalid: "xyzzy", "r2d2xxxx", "c3piozzzz"
```

If the request would, without the `If-Invalid` header, result in anything other than a 2xx status, then the `If-Invalid` header is ignored.

The purpose of this feature is to allow efficient updates of cached information with a minimum amount of transaction overhead.

10.49 If-Valid

The `If-Valid` request-header field is used with a method to make it conditional. A client that has a cache entry for the relevant entity supplies the associated validator using the `If-Valid` header; if this validator matches the server's current validator for the entity, the server SHOULD perform the requested operation as if the `If-Valid` header were not present.

If the validators do not match, the server MUST NOT perform the requested operation, and MUST return a 412 (Precondition failed) response with no `Entity-Body`. This behavior is most useful when the client wants to prevent an updating method, such as PUT or POST, from modifying a resource whose value has changed since the client last checked it.

When the `If-Valid` header is used, the server should use the strong comparison function (see section 3.13) to compare validators.

If the `If-Valid` header is used to make a conditional request on varying resource, it may be used to pass a set of validators. This is done using the variant-set mechanism if the client has variant IDs for the corresponding cache entries (see sections 13.8.3 and 3.16), or the validator-set mechanism if the client has no variant IDs (see sections 13.8.4 and 3.15).

```
If-Valid = "If-Valid" ":" if-valid-rhs
if-valid-rhs = validator-set | variant-set
```

An updating request (e.g., a PUT or POST) on a multi-entity resource should include only one variant-set-item, the one associated with the particular variant whose value is being conditionally updated.

Examples of single-entity form:

```
- If-Valid: "xyzzy"
- If-Valid: "xyzzy"/W
```

Examples of multiple-entity form:

```
- If-Valid: "xyzzy";4
- If-Valid: "xyzzy";3, "r2d2xxxx";5, "c3piozzzz";7
- If-Valid: "xyzzy", "r2d2xxxx", "c3piozzzz"
- If-Valid: "xyzzy"/W;3, "r2d2xxxx"/W;5, "c3piozzzz"/W;7
```

If the request would, without the `If-Valid` header, result in anything other than a 2xx status, then the `If-Valid` header is ignored.

The purpose of this feature is to allow efficient updates of cached information with a minimum amount of transaction overhead. It is also used, on updating requests, to prevent inadvertent modification of the wrong instance of a resource.

10.50 If-Unmodified-Since

The `If-Unmodified-Since` request-header field is used with a method to make it conditional. If the requested resource has not been modified since the time specified in this field, the server should perform the requested operation as if the `If-Unmodified-Since` header were not present.

If the requested resource has been modified since the specified time, the server MUST NOT perform the requested operation, and MUST return a 412 (Precondition failed) response with no `Entity-Body`.

```
If-Unmodified-Since = "If-Unmodified-Since" ":" HTTP-date
```

An example of the field is:

```
If-Unmodified-Since: Sat, 29 Oct 1994 19:43:31 GMT
```

If the request normally (i.e., without the `If-Unmodified-Since` header) would result in anything other than a 2xx status, the `If-Unmodified-Since` header should be ignored.

If the specified date is invalid, the header is ignored.

10.51 Warning

Warning headers are sent with responses using:

```
Warning = "Warning" ":" warn-code SP warn-agent SP warn-text
        [SP language-tag [SP charset]]
warn-code = 2DIGIT
warn-agent = ( host [ ":" port ] ) | pseudonym
                ; the name or pseudonym of the server adding
                ; the Warning header, for use in debugging
warn-text = quoted-string
```

A response may carry more than one `Warning` header.

The warn-text should be in a natural language and character set that is most likely to be intelligible to the human user receiving the response. This decision may be based on any available knowledge, such as the location of the cache or user, the `Accept-Language` field in a request, the `Content-Language` field in a response, etc. The default language is English and the default character set is ISO-8599-1.

Any server or cache may add `Warning` headers to a response. New `Warning` headers should be added after any existing `Warning` headers. A cache MUST NOT delete any `Warning` header that it received with a response. However, if a cache successfully validates a cache entry, it SHOULD remove any `Warning` headers previously attached to that entry. It MUST then add any `Warning` headers received in the validating response. In other words, `Warning` headers are those that would be attached to the most recent relevant response.

This needs clarification. Someplace else in the specification, we need to make a clear distinction between headers that are stored with a cache entry and those that aren't, and we have to define carefully what headers are simply deleted when a cache entry is updated. Section 13.7.3 already talks about combining headers, but doesn't provide a way to remove, say, a “Response is stale” Warning after a fresh response is received.

When multiple `Warning` headers are attached to a response, the user agent SHOULD display as many of them as possible, in the order that they appear in the response. If it is not possible to display all of the warnings, the user agent should follow these heuristics:

- Warnings that appear early in the response take priority over those appearing later in the response.
- Warnings in the user's preferred language and character set take priority over warnings in other languages or character sets but with identical warn-codes and warn-agents.
- TBS

This is a list of the currently-defined warn-codes, each with a recommended warn-text in English, and a description of its meaning.

10 “Response is stale”
: MUST be included whenever the returned response is stale. A cache may add this warning to any response, but may never remove it until the response is known to be fresh.

11 “Revalidation failed”
: MUST be included if a cache returns a stale response because an attempt to revalidate the response failed, due to an inability to reach the server. A cache may add this warning to any response, but may never remove it until the response is successfully revalidated.

13 “Disconnected operation”
: SHOULD be included if the cache is intentionally disconnected from the rest of the network for a period of time.

99 Miscellaneous warning
: The warning text may include arbitrary information to be presented to a human user, or logged. A system receiving this warning MUST NOT take any automated action.

TBS XXX anything else?

10.52 Vary

The `Vary` response-header field is used by an origin server to signal that the resource identified by the current request is a varying resource. A varying resource has multiple entities associated with it, all of which are representations of the content of the resource. If a GET or HEAD request on a varying resource is received, the origin server will select one of the associated entities as the entity best matching the request. Selection of this entity is based on the contents of particular header fields in the request message, or on other information pertaining to the request, like the network address of the sending client.

If a resource is varying, this has an important effect on cache management, particularly for caching proxies which service a diverse set of user agents. All 200 (OK) responses from varying resources MUST contain at least one `Vary` header or `Alternates` header (Section 10.53) to signal variance.

If no `Vary` headers and no `Alternates` headers are present in a 200 (OK) response, then caches may assume, as long as the response is fresh, that the resource in question is not varying, and has only one associated entity. Note however that this entity can still change through time, as possibly indicated by a `Cache-Control` response header (section 10.cc).

After selection of the entity best matching the current request, the origin server will usually generate a 200 (OK) response, but it can also generate other responses like 206 (Partial Content) or 304 (Not modified) if headers which modify the semantics of the request, like `Range` (Section 10.ran) or `If-Valid` (Section 10.ifva), are present. An origin server need not be capable of selecting an entity for every possible incoming request on a varying resource; it can choose to generate a 3xx (redirection) or 4xx (client error) type response for some requests.

In a request message on a varying resource, the selecting request headers are those request headers whose contents were used by the origin server to select the entity best matching the request. The Vary header field specifies the selecting request headers and any other selection parameters that were used by the origin server.

```
Vary                  = "Vary" ":" 1#selection-parameter

selection-parameter  = request-header-name
                      | "{accept-headers}"
                      | "{other}"
                      | "{" extension-parameter "}"

request-header-name  = field-name

extension-parameter  = token
```

The presence of a request-header-name signals that the request-header field with this name is selecting. Note that the name need not belong to a request-header field defined in this specification, and that header names are case-insensitive. The presence of the "{accept-headers}" parameter signals that all request headers whose names start with "accept" are selecting.

The inclusion of the "{other}" parameter in a `Vary` field signals that parameters other than the contents of request headers, for example the network address of the sending party, play a role in the selection of the response.

> Note: This specification allows the origin server to express that other parameters were used, but does not allow the origin server to specify the exact nature of these parameters. This is left to future extensions.

If an extension-parameter unknown to the cache is present in a `Vary` header, the cache MUST treat it as the "{other}" parameter. If multiple `Vary` and `Alternates` header fields are present in a response, these MUST be combined to give all selecting parameters.

The field name "`Host`" MUST never be included into a `Vary` header; clients MUST ignore it if it is present. The names of fields which change the semantics of a GET request, like "`Range`" and "`If-Valid`" MUST also never be included, and MUST be ignored when present.

Servers which use access authentication are not obliged to send "`Vary: Authorization`" headers in responses. It MUST be assumed that requests on authenticated resources can always produce different responses for different users. Note that servers can signal the absence of authentication by including a "`Cache-Control: public`" header in the response.

A cache MAY store and refresh 200 (OK) responses from a varying resource according to the rules in Section 13.7.2. The partial entities in 206 (Partial Content) responses from varying resources MAY also be used by the cache.

When getting a request on a varying resource, a cache can only return a cached 200 (OK) response to one of its clients in two particular cases.

First, if a cache gets a request on a varying resource for which it has cached one or more responses with `Vary` or `Alternates` headers, it can relay that request towards the origin server, adding an `If-Invalid` header listing the `cval-info` values in the `CVal` headers (Section 10.47) of the cached responses. If it then gets back a 304 (Not Modified) response with the `cval-info` of a cached 200 (OK) response in its `CVal` header, it can return this cached 200 (OK) response to its client, after merging in any of the 304 response headers as specified in Section 13.7.2.

Second, if a cache gets a request on a varying resource, it can return to its client a cached, fresh 200 (OK) response which has `Vary` or `Alternates` headers, provided that

- the `Vary` and `Alternates` headers of this fresh response specify that only request header fields are selecting parameters,
- the specified selecting request header fields of the current request match the specified selecting request header fields of a previous request on the resource relayed towards the origin server,
- this previous request got a 200 (OK) or 304 (Not Modified) response which had the same `cval-info` value in its `CVal` header as the cached, fresh 200 (OK) response.

Two sequences of selecting request header fields match if and only if the first sequence can be transformed into the second sequence by only adding or removing whitespace at places in fields where this is allowed according to the syntax rules in this specification.

If a cached 200 (OK) response MAY be returned to a request on a varying resource which includes a `Range` request header, then a cache MAY also use this 200 (OK) response to construct and return a 206 (Partial Content) response with the requested range.

> Note: Implementation of support for the second case above is mainly interesting in user agent caches, as a user agent cache will generally have an easy way of determining whether the sequence of request header fields of the current request equals the sequence sent in an earlier request on the same resource. Proxy caches supporting the second case would have to record diverse sequences of request header fields previously relayed; the implementation effort associated with this may not be balanced by a sufficient payoff in traffic savings. A planned specification of a content negotiation mechanism will define additional cases in which proxy caches can return a cached 200 (OK) response without contacting the origin server. The implementation effort associated with support for these additional cases is expected to have a much better cost/benefit ratio.

10.53 Alternates

The `Alternates` response-header field is used by origin servers to signal that the resource identified by the current request has the capability to send different responses depending on the accept headers in the request message. This has an important effect on cache management, particularly for caching proxies which service a diverse set of user agents. This effect is covered in Section 10.v.

```
Alternates          = "Alternates" ":" opaque-field

opaque-field        = field-value
```

The `Alternates` header is included into HTTP/1.1 to make HTTP/1.1 caches compatible with a planned content negotiation mechanism. HTTP/1.1 allows a future content negotiation standard to define the format of the `Alternates` header field-value, as long as the defined format satisfies the general rules in Section 4.2.

To ensure compatibility with future experimental or standardized software, caching HTTP/1.1 clients MUST treat all `Alternates` headers in a response as synonymous to the following `Vary` header:

```
Vary: {accept-headers}
```

and follow the caching rules associated with the presence of this `Vary` header, as covered in Section 10.v. HTTP/1.1 allows origin servers to send `Alternates` headers under experimental conditions.

10.54 SLUSHY: Accept-Ranges

In some cases, a client may want to know if the server accepts range requests using a certain range unit. The server may indicate its acceptance of range requests for a resource by providing this header in a response for that resource:

```
Accept-Ranges     = "Accept-Ranges" ":" acceptable-ranges
acceptable-ranges = 1#range-unit | "none"
```

Origin servers that accept byte-range requests MAY send

```
Accept-Ranges: bytes
```

but are not required to do so. Clients MAY generate byte-range requests without having received this header for the specific resource involved, but the server MAY ignore such requests.

Should this say that the server SHOULD send "Accept-Ranges: bytes", or is MAY good enough

Origin servers that do not accept any kind of range request for a specific resource MAY send

```
Accept-Ranges: none
```

to advise the client not to attempt a range request.

We're still not quite sure why this header is in the protocol. We gather that Netscape uses it for something, but nobody from Netscape has even tried to explain to me whether it is necessary for anything. The only thing we can think of is that a client would have to know in advance if a server accepted partial-content PUTs (i.e., PUT+Content-Range), but we don't see any indication that this is what Netscape wants.

10.55 SLUSHY: Range-If

If a client has a partial copy of an entity in its cache, and wished to have an up-to-date copy of the entire entity in its cache, it could use `Range` request header with a conditional GET (using either of both of `If-Unmodified-Since` and `If-Valid`.) However, if the condition fails because the entity has been modified, the client would then have to make a second request to obtain the entire current entity body.

The `Range-If` header allows a client to ``short-circuit'' the second request. Informally, its meaning is ``if the entity is unchanged, send me the part(s) that I am missing; otherwise, send me the entire new entity.''

```
Range-If: if-valid-rhs
```

The `Range-If` header should only be used together with a `Range` header,and must be ignored if the request does not include a `Range` header, or if the server does not support the sub-range operation.

If the validator given in the `Range-If` header matches the current validator for the entity, then the server should provide the specified sub-range of the entity using a 206 (Partial content) response. If the validator does not match, then the server should return the entire entity using a 200 (OK) response.

This description may need slight modification to deal with(1) the use of a last-modified date as a validator (but this |can perhaps be hidden in the definition of if-valid-rhs), and|(2) its application to multi-entity resources.

11. Access Authentication

HTTP provides a simple challenge-response authentication mechanism which MAY be used by a server to challenge a client request and by a client to provide authentication information. It uses an extensible, case-insensitive token to identify the authentication scheme, followed by a comma-separated list of attribute-value pairs which carry the parameters necessary for achieving authentication via that scheme.

```
auth-scheme    = token

auth-param     = token "=" quoted-string
```

The 401 (unauthorized) response message is used by an origin server to challenge the authorization of a user agent. This response MUST include a `WWW-Authenticate` header field containing at least one `challenge` applicable to the requested resource.

```
challenge      = auth-scheme 1*SP realm *( "," auth-param )

realm          = "realm" "=" realm-value
realm-value    = quoted-string
```

The realm attribute (case-insensitive) is required for all authentication schemes which issue a challenge. The realm value (case-sensitive), in combination with the canonical root URL of the server being accessed, defines the protection space. 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 may have additional semantics specific to the authentication scheme.

A user agent that wishes to authenticate itself with a server--usually, but not necessarily, after receiving a 401 or 411 response--MAY do so by including an `Authorization` header field with the request. The `Authorization` field value consists of `credentials` containing the authentication information of the user agent for the realm of the resource being requested.

```
credentials    = basic-credentials
               | auth-scheme *("," auth-param )
```

The domain over which credentials can be automatically applied by a user agent is determined by the protection space. If a prior request has been authorized, the same credentials MAY be reused for all other requests within that protection space for a period of time determined by the authentication scheme, parameters, and/or user preference. Unless otherwise defined by the authentication scheme, a single protection space cannot extend outside the scope of its server.

If the server does not wish to accept the credentials sent with a request, it SHOULD return a 401 (unauthorized) response. The response MUST include a `WWW-Authenticate` header field containing the (possibly new) `challenge` applicable to the requested resource and an entity explaining the refusal.

The HTTP protocol does not restrict applications to this simple challenge-response mechanism for access authentication. Additional mechanisms MAY be used, such as encryption at the transport level or via message encapsulation, and with additional header fields specifying authentication information. However, these additional mechanisms are not defined by this specification.

Proxies MUST be completely transparent regarding user agent authentication. That is, they MUST forward the `WWW-Authenticate` and `Authorization` headers untouched, and MUST not cache the response to a request containing `Authorization`.

HTTP/1.1 allows a client to pass authentication information to and from a proxy via the `Proxy-Authenticate` and `Proxy-Authorization` headers.

11.1 Basic Authentication Scheme

The “basic” authentication scheme is based on the model that the user agent must authenticate itself with a user-ID and a password for each realm. The realm value should be considered an opaque string which can only be compared for equality with other realms on that server. The server will service the request only if it can validate the user-ID and password for the protection space of the `Request-URI`. There are no optional authentication parameters.

Upon receipt of an unauthorized request for a URI within the protection space, the server SHOULD respond with a challenge like the following:

```
WWW-Authenticate: Basic realm="WallyWorld"
```

where “WallyWorld” is the string assigned by the server to identify the protection space of the `Request-URI`.

To receive authorization, the client sends the user-ID and password, separated by a single colon (“:”) character, within a base64 [7] encoded string in the `credentials`.

```
basic-credentials = "Basic" SP basic-cookie

basic-cookie   = <base64 [7] encoding of userid-password,
                  except not limited to 76 char/line>

userid-password = [ token ] ":" *TEXT
```

If the user agent wishes to send the user-ID “Aladdin” and password “open sesame”, it would use the following header field:

```
Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
```

The basic authentication scheme is a non-secure method of filtering unauthorized access to resources on an HTTP server. It is based on the assumption that the connection between the client and the server can be regarded as a trusted carrier. As this is not generally true on an open network, the basic authentication scheme should be used accordingly. In spite of this, clients SHOULD implement the scheme in order to communicate with servers that use it.

11.2 Digest Authentication Scheme

The “digest” authentication scheme is [currently described in an expired Internet-Draft, and this description will have to be improved to reference a new draft or include the old one].

12. Content Negotiation

A varying resource has multiple entities associated with it, all of which are representations of the content of the resource. Content negotiation is the process of selecting the best representation when a GET or HEAD request is made on the varying resource. HTTP/1.1 has provisions for two kinds of content negotiation: opaque negotiation and transparent negotiation.

With opaque negotiation, the selection of the best representation is done by an algorithm located at the origin server, and unknown to the proxies and user agents involved. Selection is based on the contents of particular header fields in the request message, or on other information pertaining to the request, like the network address of the sending client. A typical example of opaque negotiation would be the selection of a text/html response in a particular language based on the contents of the `Accept-Language` request header field. A disadvantage of opaque negotiation is that the request headers may not always contain enough information to allow for selection. If the `Accept` header

```
Accept: text/*: q=0.3, text/html, */*: q=0.5
```

is sent in a request on a varying resource which has a video/mpeg and a video/quicktime representation, the selection algorithm in the origin server will either have to make a default choice, or return an error response which allows the user to decide on further actions.

With transparent negotiation, the selection of the best representation is done by a distributed algorithm which can perform computation steps in the origin server, in proxies, or in the user agent. Transparent negotiation guarantees that, if the user agent supports the transparent negotiation algorithm and is correctly configured, the request will always correctly yield either the video/mpeg representation, the video/quicktime representation, or an error message indicating that the resource cannot be displayed by the user agent.

12.1 Negotiation facilities defined in this specification

This specification defines all protocol facilities for opaque negotiation, but does not define the distributed algorithm for transparent negotiation. This specification only defines the basic facilities (`Vary`, `Alternates`, `Accept`) in the core protocol allowing requests on transparently negotiated resources to be correctly handled by HTTP/1.1 caches. All other information about transparent content negotiation is found in a separate document[29].

If a varying resource is opaquely negotiated, successful responses to requests on the resource will always include a `Vary` header. If a varying resource is transparently negotiated, successful responses to requests on the resource will always include an `Alternates` header. If a successful response contains an `Al`ternates header, it will also always contain a `Content-Location` header. A future specification may allow a combination of opaque and transparent negotiation that would lead to the inclusion of both a `Vary` header and an `Alternates` header in a response.

.

13 Caching in HTTP

The World Wide Web is a distributed system, and so its performance can be improved by the use of caches. These caches are typically placed at proxies and in the clients themselves. The HTTP/1.1 protocol includes a number of elements intended to make caching work as well as possible. Because these elements are inextricable from other aspects of the protocol, and because they interact with each other, it is useful to describe the basic caching design of HTTP separately from the detailed descriptions of methods, headers, response codes, etc.

13.1 Semantic Transparency

Ideally, an HTTP/1.1 cache would be “semantically transparent.” That is, use of the cache would not affect either the clients or the servers in any way except to improve performance. When a client makes a request via a semantically transparent cache, it receives exactly the same entity headers and entity body it would have received if it had made the same request to the origin server, at the same time.

In the real world, requirements for performance, availability, and disconnected operation require us to relax the goal of semantic transparency in many cases. The HTTP/1.1 protocol allows origin servers, caches, and clients to explicitly reduce transparency when necessary. However, because non-transparent operation may confuse non-expert users, and may be incompatible with certain server applications (such as those for ordering merchandise), the protocol requires that transparency may not be relaxed

- without an explicit protocol-level request (when relaxed by client or origin server)
- without a means for warning the end user (when relaxed by cache or client)

Therefore, the HTTP/1.1 protocol provides these important elements:

1. Protocol features that provide full semantic transparency when this is desired by all parties.
2. Protocol features that allow an origin server or end-user client to explicitly request and control non-transparent operation.
3. Protocol features that allow a cache to attach warnings to responses that do not preserve semantic transparency.

A basic principle is that it must be possible for the clients to detect any potential breakdown of semantic transparency.

Caching would be useless if it did not significantly improve performance in many cases. The goal of caching in HTTP/1.1 is to eliminate the need to send requests in many cases, and to eliminate the need to send full responses in many other cases. The former reduces the number of network round-trips required for many operations; we use an “expiration” mechanism for this purpose (see section 13.2). The latter reduces network bandwidth requirements; we use a “validation” mechanism for this purpose (see section 13.3).

The server, cache, or client implementer may be faced with design decisions not explicitly discussed in this specification. If decision may affect semantic transparency, the implementer ought to err on the side of maintaining transparency unless a careful and complete analysis shows significant benefits in breaking transparency.

A note on terminology: we say that a resource is “cachable” if a cache is allowed to store a copy of this resource, when it arrives in a response message, and then later use that copy to respond to a subsequent request. Even if a resource is cachable, there may be additional constraints on when and whether a cache can use a cached copy of it.

13.2 Expiration Model

In order to describe the associated mechanisms, we introduce several terms for describing responses returned by a cache in response to a client's request:

firsthand
: A response is firsthand if it comes directly and without unnecessary delay from the origin server, perhaps via one or more proxies. A response is also firsthand if its validity has just been checked directly with the origin server.

explicit expiration time
: The time at which the origin server intends that an entity should no longer be returned by a cache without further validation.

heuristic expiration time
: An expiration time assigned by a cache when no explicit expiration time is available.

age
: The age of a response is the time since it was generated by, or successfully validated with, the origin server.

freshness lifetime
: The length of time between the generation of a response and its expiration time.

fresh
: A response is fresh if its age has not yet reached its freshness lifetime.

stale

A response is stale if its age has passed its freshness lifetime.

13.2.1 Server-Specified Expiration

HTTP caching works best when caches can entirely avoid making requests to the origin server. The primary mechanism for avoiding requests is for an origin server to provide an explicit expiration time in the future, indicating that a response may be used to satisfy subsequent requests. In other words, a cache can return a fresh response without first contacting the server.

Our expectation is that servers will assign future explicit expiration times to responses in the belief that the entity is not likely to change, in a semantically significant way, before the expiration time is reached. This normally preserves semantic transparency, as long as the server's expiration times are carefully chosen.

If an origin server wishes to force a semantically transparent cache to validate every request, it may assign an explicit expiration time in the past. This means that the response is always stale, and so the cache SHOULD validate it before using it for subsequent requests. (Note that a firsthand response MUST always be returned to the requesting client, independent of its expiration time, unless the connection to the client is lost.)

If an origin server wishes to force any HTTP/1.1 cache, no matter how it is configured, to validate every request, it should use the "`must-revalidate`" `Cache-Control` directive (see section 10.8).

Servers specify explicit expiration times using either the `Expires` header, or the `max-age` directive of the `Cache-Control` header.

13.2.2 Limitations on the Effect of Expiration Times

An expiration time cannot be used to force a user agent to refresh its display or reload a resource; its semantics apply only to caching mechanisms, and such mechanisms need only check a resource's expiration status when a new request for that resource is initiated.

User agents often have history mechanisms, such as "Back" buttons and history lists, which can be used to redisplay an entity retrieved earlier in a session. By default, an expiration time does not apply to history mechanisms. If the entity is still in storage, a history mechanism should display it even if the entity has expired, unless the user has specifically configured the agent to refresh expired history documents.

13.2.3 Heuristic Expiration

Since origin servers do not always provide explicit expiration times, HTTP caches typically assign heuristic expiration times, employing algorithms that use other header values (such as the `Last-Modified` time) to estimate a plausible expiration time. The HTTP/1.1 specification does not provide specific algorithms, but does impose worst-case constraints on their results. Since heuristic expiration times may compromise semantic transparency, they should be used cautiously, and we encourage origin servers to provide explicit expiration times as much as possible.

13.2.4 Client-controlled Behavior

While the origin server (and to a lesser extent, intermediate caches) are the primary source of expiration information, in some cases the client may need to control a cache's decision about whether to return a cached response without validating it. Clients do this using several directives of the `Cache-Control` header.

A client's request may specify the maximum age it is willing to accept for an unvalidated response; specifying a value of zero forces the cache(s) to revalidate all responses. A client may also specify the minimum time remaining before a response expires. Both of these options increase constraints on the behavior of caches, and so cannot decrease semantic transparency.

A client may also specify that it will accept stale responses, up to some maximum amount of staleness. This loosens the constraints on the caches, and so may violate semantic transparency, but may be necessary to support disconnected operation, or high availability in the face of poor connectivity.

13.2.5 Exceptions to the Rules and Warnings

In some cases, the operator of a cache may choose to configure it to return stale responses even when not requested by clients. This decision not be made lightly, but may be necessary for reasons of availability or performance, especially when the cache is poorly connected to the origin server. Whenever a cache returns a stale response, it MUST mark it as such (using a `Warning` header). This allows the client software to alert the user that there may be a potential problem.

It also allows the user to take steps to obtain a firsthand or fresh response, if the user so desires. For this reason, a cache MUST NOT return a stale response if the client explicitly requests a first-hand or fresh one, unless it is impossible to comply.

13.2.6 Age Calculations

In order to know if a cached entry is fresh, a cache needs to know if its age exceeds its freshness lifetime. We discuss how to calculate the latter in section 13.2.7; this section describes how to calculate the age of a response or cache entry.

In this discussion, we use the term “now” to mean “the current value of the clock at the host performing the calculation.” All HTTP implementations, but especially origin servers and caches, should use NTP [RFC1305] or some similar protocol to synchronize their clocks to a globally accurate time standard.

Also note that HTTP/1.1 requires origin servers to send a `Date` header with every response, giving the time at which the response was generated. We use the term “date_value” to denote a representation of the value of the `Date` header, in a form appropriate for arithmetic operations.

HTTP/1.1 uses the “`Age`” response header to help convey age information between caches. The `Age` header value is the sender's estimate of the amount of time since the response was generated at the origin server. In the case of a cached response that has been revalidated with the origin server, the `Age` value is based on the time of revalidation, not of the original response.

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.

We use the term “age_value” to denote a representation of the value of the `Age` header, in a form appropriate for arithmetic operations.

An response's age can be calculated in two entirely independent ways:

1. now - date_value, if the local clock is reasonably well synchronized to the origin server's clock. If the result is negative, this is replaced by zero.
2. age_value, if all of the caches along the response path implement HTTP/1.1.

Given that we have two independent ways to compute the age of a response when it is received, we can combine these as

```
corrected_received_age = max(now - date_value, age_value)
```

and as long as we have either nearly synchronized clocks or all-HTTP/1.1 paths, one gets a reliable (conservative) result.

Note that this correction is applied at each HTTP/1.1 cache along the path, so that if there is an HTTP/1.0 cache in the path, the correct received age is computed as long as the receiving cache's clock is nearly in sync. We don't need end-to-end clock synchronization (although it is good to have), and there is no explicit clock synchronization step.

Because of network-imposed delays, some significant interval may pass from the time that a server generates a response, and the time it is received at the next outbound cache or client. If uncorrected, this delay could result in improperly low ages.

Because the request that resulted in the returned `Age` value must have been initiated prior to that `Age` value's generation, we can correct for delays imposed by the network by recording the time at which the request was initiated. Then, when an `Age` value is received, it MUST be interpreted relative to the time the request was initiated, not the time that the response was received. This algorithm results in conservative behavior no matter how much delay is experienced. So, we compute:

```
corrected_initial_age = corrected_received_age + (now - request_time)
```

where “request_time” is the time (according to the local clock) when the request that elicited this response was sent.

Summary of age calculation algorithm, when a cache receives a response:

```
/*
 * age_value
 *      is the value of Age: header received by the cache with
 *              this response.
 * date_value
 *      is the value of the origin server's Date: header
 * request_time
 *      is the (local) time when the cache made the request
 *              that resulted in this cached response
 * response_time
 *      is the (local) time when the cache received the
 *              response
 * now
 *      is the current (local) time
 */
apparent_age = max(0, now - date_value);
corrected_received_age = max(apparent_age, age_value);
response_delay = now - request_time;
corrected_initial_age = corrected_received_age + response_delay;
resident_time = now - response_time;
current_age = corrected_initial_age + resident_time;
```

When a cache sends a response, it must add to the corrected_initial_age the amount of time that the response was resident locally. It must then transmit this total age, using the Age header, to the next recipient cache.

13.2.7 Expiration Calculations

In order to decide whether a response is fresh or stale, we need to compare its freshness lifetime to its age. The age is calculated as described in section 13.2.6; this section describes how to calculate the freshness lifetime, and to determine if a response has expired.

We use the term “expires_value” to denote a representation of the value of the Expires header, in a form appropriate for arithmetic operations. We use the term “max_age_value” to denote an appropriate representation of the number of seconds carried by the max-age directive of the Cache-Control header in a response (see section 10.8).

The max-age directive takes priority over Expires, so if max-age is present in a response, the calculation is simply:

```
freshness_lifetime = max_age_value
```

Otherwise, if `Expires` is present in the response, the calculation is:

```
freshness_lifetime = expires_value - date_value
```

Note that neither of these calculations is vulnerable to clock skew, since all of the information comes from the origin server.

If neither Expires nor Cache-Control max-age appears in the response, and the response does not include other restrictions on caching, the cache MAY compute a freshness lifetime using a heuristic. This heuristic is subject to certain limitations; the minimum value may be zero, and the maximum value MUST be no more than 24 hours.

Also, if the response does have a `Last-Modified` time, the heuristic expiration value SHOULD be no more than some fraction of the interval since that time. A typical setting of this fraction might be 10%.

The calculation to determine if a response has expired is quite simple:

```
response_is_fresh = (freshness_lifetime > current_age)
```

13.2.8 UT Mandatory

All expiration-related calculations must be done in Universal Time (GMT). The local time zone MUST not influence the calculation or comparison of an age or expiration time.

If an HTTP header incorrectly carries a date value with a time zone other than GMT, it must be converted into GMT using the most conservative possible conversion.

13.3 Validation Model

When a cache has a stale value that it would like to use as a response to a client's request, it first has to check with the origin server (or possibly an intermediate cache with a fresh response) to see if its cached value is still usable. We call this "validating" the cache entry. Since we do not want to have to pay the overhead of retransmitting the full response if the cached value is good, and we do not want to pay the overhead of an extra round trip if the cached value is invalid, the HTTP/1.1 protocol supports the use of conditional methods.

The key protocol features for supporting conditional methods are those concerned with "cache validators." When an origin server generates a full response, it attaches some sort of validator to it, which is kept with the cache entry. When a client (end-user or cache) makes a conditional request for a resource for which it has a cache entry, it includes the associated validator in the request.

The server then checks that validator against the current validator for the resource, and if they match, it responds with a special status code (usually, "304 Not Modified") and no entity body. Otherwise, it returns a full response (including entity body). Thus, we avoid transmitting the full response if the validator matches, and we avoid an extra round trip if it does not match.

> Note: the comparison functions used to decide if validators match are defined in section 13.3.3.

In HTTP/1.1, a conditional request looks exactly the same as a normal request for the same resource, except that it carries a special header (which includes the validator) that implicitly turns the method (usually, GET) into a conditional.

The protocol includes both positive and negative senses of cache-validating conditions. That is, it is possible to request either that a method be performed if and only if the validators match, or if and only if the validators do not match.

> Note: a response that lacks a cache validator may still be cached, and served from cache until it expires, unless this is explicitly prohibited by a `Cache-Control` directive. However, a cache cannot do a conditional retrieval if it does not have a cache validator for the entity, which means it will not be refreshable after it expires.

13.3.1 Last-modified Dates

In HTTP/1.0, the only cache validator is the `Last-Modified` time carried by a response. Clients validate entities using the `If-Modified-Since` header. In simple terms, a cache entry is considered to be valid if the actual resource has not been modified since the original response was generated.

13.3.2 Opaque Validators

HTTP/1.1 introduces the possibility of using an "opaque" validator, for situations where the `Last-Modified` date is not appropriate. This may include server implementations where it is not convenient to store modification dates, or where the one-second resolution of HTTP date values is insufficient, or where the origin server wishes to avoid certain paradoxes that may arise from the use of modification dates.

An opaque validator is simply a string of octets whose internal structure is not known to clients or caches. Caches store opaque validators and return them when making conditional requests. Also, when a cache receives a conditional request for a resource for which it has a fresh cache entry, it may compare opaque validators using strict octet-equality. Otherwise, opaque validators have no semantic value to clients or caches.

To preserve compatibility with HTTP/1.0 clients and caches, and because the `Last-Modified` date may be useful for purposes other than cache validation, HTTP/1.1 servers SHOULD send `Last-Modified` whenever feasible.

The headers used to convey opaque validators are described in sections 10.47, 10.48, 10.49, and 10.55.

13.3.3 Weak and Strong Validators

Since both origin servers and caches will compare two validator values to decide if they represent the same or different values for the entire resource, one normally would expect that if the resource value (the entity body or any entity headers) changes in any way, then the associated validator would change as well. If this is true, then we call this validator a "strong validator."

However, there may be cases when a server prefers to change the validator only on semantically significant changes, and not when insignificant aspects of the resource change. A validator that does not always change when the resource changes is a "weak validator."

One can think of a strong validator as one that changes whenever the bits of an entity changes, while a weak value changes whenever the meaning of an entity changes. Alternatively, one can think of a strong validator as part of an identifier for a specific instance of an entity, while a weak validator is part of an identifier for a set of semantically equivalent instances of an entity.

> Note: One example of a strong validator is an integer that is incremented in stable storage every time an entity is changed.
>
> An entity's modification time, if represented with one-second resolution, could be a weak validator, since it is possible that the resource may be modified twice during a single second.

Opaque validators are normally "strong," but the protocol provides a mechanism to tag an opaque validator as "weak."

A "use" of a validator is either when a client generates a request and includes the validator in a validating header field, or when a server compares two validators.

Strong validators are usable in any context. Weak validators are only usable in contexts that do not depend on exact equality of an entity. For example, either kind is usable for a conditional GET of a full entity. However, only a strong validator is usable for a sub-range retrieval, since otherwise the client may end up with an internally inconsistent entity body.

The only function that the HTTP/1.1 protocol defines on validators is comparison. There are two validator comparison functions, depending on whether the comparison context allows the use of weak validators or not:

- The strong comparison function: in order to be considered equal, both validators must be identical in every way, and neither may be weak.
- The weak comparison function: in order to be considered equal, both validators must be identical in every way, but either or both of them may be tagged as "weak" without affecting the result.

The weak comparison function should be used for simple (non-subrange) GET requests. The strong comparison function must be used in all other cases.

An opaque validator is strong unless it is explicitly tagged as weak. Section 3.13 gives the syntax for opaque validators.

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:

- The validator is being compared by an origin server to the actual current validator for the entity and,
- That origin server reliably knows that the associated entity did not change twice during the second covered by the presented validator.

or

- The validator is about to be used by a client in an `If-Modified-Since` or `If-Unmodified-Since` header, because the client has a cache entry for the associated entity, and
- That cache entry include a `Date` value, which gives the time when the origin server generated the original response, and
- The presented `Last-Modified` time is at least 60 seconds before the `Date` value.

or

- The validator is being compared by an intermediate cache to the validator stored in its cache entry for the entity, and
- That cache entry include a `Date` value, which gives the time when the origin server generated the original response, and
- 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 generated 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.

If a client wishes to perform a sub-range retrieval on a value for which it has only a `Last-Modified` time and no opaque validator, it may do this only if the `Last-Modified` time is strong in the sense described here.

A cache or origin server receiving a cache-conditional request, other than a full-body GET request, must use the strong comparison function to evaluate the condition.

This allows HTTP/1.1 caches and clients to safely perform sub-range retrievals on values that have been obtained from HTTP/1.0 servers.

13.3.4 Rules for When to Use Opaque Validators and Last-modified Dates

We adopt a set of rules and recommendations for origin servers, clients, and caches regarding when various validator types should be used, and for what purposes.

HTTP/1.1 origin servers:

- SHOULD send a strong opaque validator unless performance considerations support the use of weak opaque validators, or unless it is unfeasible to send a strong opaque validator.
- MAY send a weak opaque validator instead of a strong one.
- MAY send no opaque validator if it is infeasible to generate one.
- SHOULD send a `Last-Modified` value if it is feasible to send one, unless the risk of a breakdown in semantic transparency that could result from using this date in an `If-Modified-Since` header would lead to serious problems.

In other words, the preferred behavior for an HTTP/1.1 origin server is to send both a strong opaque validator and a `Last-Modified` value.

In order to be legal, a strong opaque validator MUST change whenever the associated entity value changes in any way. A weak opaque validator SHOULD change whenever the associated entity value changes in a semantically significant way.

> Note: in order to provide semantically transparent caching, an origin server should avoid reusing a specific strong opaque validator value for two different instances of an entity, or reusing a specific weak opaque validator value for two semantically different instances of an entity. Caches entries may persist for arbitrarily long periods, regardless of expiration times, so it may be inappropriate to expect that a cache will never again attempt to validate an entry using a validator that it obtained at some point in the past.

HTTP/1.1 clients:

- If an opaque validator has been provided by the origin server, MUST use that validator in any cache-conditional request (using `If-Valid` or `If-Invalid`).
- If only a `Last-Modified` value has been provided by the origin server, SHOULD use that value in non-subrange cache-conditional requests (using `If-Modified-Since`).
- If only a `Last-Modified` value has been provided by an HTTP/1.0 origin server, MAY use that value in subrange cache-conditional requests (using `If-Unmodified-Since`:). The user agent should provide a way to disable this, in case of difficulty.
- If both an opaque validator and a `Last-Modified` value have been provided by the origin server, SHOULD use both validators in cache-conditional requests. This allows both HTTP/1.0 and HTTP/1.1 caches to respond appropriately.

An HTTP/1.1 cache, upon receiving a request, MUST use the most restrictive validator when deciding whether the client's cache entry matches the cache's own cache entry. This is only an issue when the request contains both an opaque validator and a last-modified-date validator (`If-Modified-Since` or `If-Unmodified-Since`:).

> A note on rationale: The general principle behind these rules is that HTTP/1.1 servers and clients should transmit as much non-redundant information as is available in their responses and requests. HTTP/1.1 systems receiving this information will make the most conservative assumptions about the validators they receive.
>
> HTTP/1.0 clients and caches will ignore opaque validators. Generally, last-modified values received or used by these systems will support transparent and efficient caching, and so HTTP/1.1 origin servers should provide `Last-Modified` values. In those rare cases where the use of a `Last-Modified` value as a validator by an HTTP/1.0 system could result in a serious problem, then HTTP/1.1 origin servers should not provide one.

13.3.5 SLUSHY: Non-validating conditionals

TBS

The principle behind opaque validators is that only the service author knows the semantics of a resource well enough to select an appropriate cache validation mechanism, and the specification of any validator comparison function more complex than byte-equality would open up a can of worms. Thus, comparisons of any other headers (except `Last-Modified`, for compatibility with HTTP/1.0) are never used for purposes of validating a cache entry.

13.3.6 FLUID: Other Issues

TBS: what if no validator present in response?

13.4 Cache-control Mechanisms

The basic cache mechanisms in HTTP/1.1 (server-specified expiration times and validators) are implicit directives to caches. In some cases, a server or client may need to provide explicit directives to the HTTP caches. We use the `Cache-Control` header for this purpose.

The `Cache-Control` header allows a client or server to transmit a variety of directives in either requests or responses. These directives typically override the default caching algorithms. As a general rule, if there is any apparent conflict between header values, the most restrictive interpretation should be applied (that is, the one that is most likely to preserve semantic transparency). However, in some cases, `Cache-Control` directives are explicitly specified as weakening semantic transparency (for example, “`max-stale`” or “`public`”).

The `Cache-Control` directives are described in detail in section 10.7.

13.5 Warnings

Whenever a cache returns a response that is not semantically transparent, it must attach a warning to that effect, using a `Warning` response header. This warning allows clients and user agents to take appropriate action.

Warnings may be used for other purposes, both cache-related and otherwise. The use of a warning, rather than an error status code, distinguish these responses from true failures.

Warnings are always cachable, because they never weaken the transparency of a response. This means that warnings can be passed to HTTP/1.0 caches without danger; such caches will simply pass the warning along as a entity header in the response.

Warnings are assigned numbers between 0 and 99. This specification defines the code numbers and meanings of each warning, allowing a client or cache to take automated action in some (but not all) cases.

Warnings also carry a warning message text in any appropriate natural language (perhaps based on the client's `Accept` headers), and an optional indication of what language and character set are used.

Multiple warning messages may be attached to a response (either by the origin server or by a cache), including multiple warnings with the same code number. For example, a server may provide the same warning with texts in both English and Basque.

When multiple warnings are attached to a response, it may not be practical or reasonable to display all of them to the user. This version of HTTP does not specify strict priority rules for deciding which warnings to display and in what order, but does suggest some heuristics.

The `Warning` header and the currently defined warnings are described in section 10.106.

13.6 Explicit Indications Regarding User-specified Overrides

Many user agents make it possible for users to override the basic caching mechanisms. For example, the user agent may allow the user to specify that cached entities (even explicitly stale ones) are never validated. Or the user agent might habitually add “`Cache-Control`: max-stale=3600” or “`Cache-Control`: reload” to every request. We recognize that there may be situations which require such overrides, although user agents SHOULD NOT default to any behavior contrary to the HTTP/1.1 specification. That is, the user should have to explicitly request either non-transparent behavior, or behavior that results in abnormally ineffective caching.

If the user has overridden the basic caching mechanisms, the user agent should explicitly indicate to the user whenever this results in the display of information that might not meet the server's transparency requirements (in particular, if the displayed resource is known to be stale). Since the protocol normally allows the user agent to determine if responses are stale or not, this indication need only be displayed when this actually happens. The indication need not be a dialog box; it could be an icon (for example, a picture of a rotting fish) or some other visual indicator.

If the user has overridden the caching mechanisms in a way that would abnormally reduce the effectiveness of caches, the user agent should continually display an indication (for example, a picture of currency in flames) so that the user does not inadvertently consume excess resources or suffer from excessive latency.

13.7 Constructing Responses From Caches

The purpose of an HTTP cache is to store information received in response to requests, for use in responding to future requests. In many cases, a cache simply returns the appropriate parts of a response to the requester. However, if the cache holds a cache entry based on a previous response, it may have to combine parts of a new response with what is held in the cache entry.

13.7.1 End-to-end and Hop-by-hop Headers

For the purpose of defining the behavior of caches and non-caching proxies, we divide HTTP headers into two categories:

- End-to-end headers, which must be transmitted to the ultimate recipient of a request or response. End-to-end headers in responses must be stored as part of a cache entry and transmitted in any response formed from a cache entry.

- Hop-by-hop headers, which are meaningful only for a single transport-level connection, and are not stored by caches or forwarded by proxies.

The following HTTP/1.1 headers are hop-by-hop headers:

- `Connection`
- `Keep-Alive`
- `Upgrade`
- `Public`
- `Proxy-Authenticate`
- `Transfer-Encoding`

All other headers defined by HTTP/1.1 are end-to-end headers.

Hop-by-hop headers introduced in future versions of HTTP MUST be listed in a `Connection` header, as described in section 10.9.

13.7.2 Non-modifiable Headers

Some features of the HTTP/1.1 protocol, such as Digest Authentication (see TBS), depend on the value of certain end-to-end headers. A cache or non-caching proxy `SHOULD NOT` modify an end-to-end header unless the definition of that header requires or specifically allows that.

A cache or non-caching proxy MUST NOT modify any of the following fields in a request or response, nor may it add any of these fields if not already present:

- `Content-Type`
- `Content-Encoding`
- `Content-Length`
- `Expires`
- `Last-Modified`
- `Content-Range`
- `Content-Location`

Warning: unnecessary modification of end-to-end headers may cause authentication failures if stronger authentication mechanisms are introduced in later versions of HTTP. Such authentication mechanisms may rely on the values of header fields not listed here.

13.7.3 Combining Headers

When a cache makes a validating request to a server, and the server provides a 304 Not Modified response, the cache must construct a response to send to the requesting client. The cache uses the entity-body stored in the cache entry as the entity-body of this outgoing response. It uses the end-to-end headers from the incoming response, not from the cache entry. Unless it decides to remove the cache entry, it must also replace the end-to-end headers stored with the cache entry with those received in the incoming response.

In other words, the complete set of end-to-end headers received in the incoming response overrides all end-to-end headers stored with the cache entry. The cache may add `Warning` headers (see section 10.106) to this set.

A cache MUST preserve the order of all headers as received in an incoming response.

These rule allows an origin server to completely control the response seen by the client of a cache when the cache revalidates an entry, and may be necessary for preserving semantic transparency or for certain kinds of security mechanisms or future extensions.

13.7.4 Combining Byte Ranges

A response may transfer only a subrange of the bytes of an entity, either because the request included one or more `Range` specifications, or because a connection was broken prematurely. After several such transfers, a cache may have received several ranges of the same entity.

If a cache has a stored non-empty set of subranges for an entity, and an incoming response transfers another subrange, the cache MAY combine the new subrange with the existing set if both the following conditions are met:

- Both the incoming response and the cache entry must have a cache validator.
- The two validators must match using the strong comparison function (see section 13.3.3).

If either requirement is not meant, the cache must use only the most recent partial response (based on the `Date` values transmitted with every response, and using the incoming response if these values are equal or missing), and must discard the other partial information.

13.7.5 SLUSHY: Scope of Expiration

HTTP/1.1's expiration model is that as soon as any variant of a URI becomes stale, all variants becomes stale as well. Thus, "freshness" applies to all the variants of URI, rather than any particular variant. Dates and expires etc. apply to any cached variant that a proxy might have with a URI and not just the one particular entity.

13.8 Caching and Content Negotiation

The HTTP content negotiation mechanism interacts with caching in several ways:

- A varying resource (one subject to content negotiation) may be bound to more than one entity. Each of these entities is called a "variant" of the resource.
- The request-URI may be only one part of the cache key.

13.8.1 Use of the Vary header

Origin servers may respond to requests for varying resources use the `Vary` header (see section 10.vary for a full description) to inform the cache which header fields of the request were used to select the variant returned in the response. A cache can use that response to reply to a subsequent request only if the two requests not only specify the same URI, but also have the same value for all headers specified in the Vary response-header.

The `Vary` header may also inform the cache that the variant was selected using criteria not limited to the request headers; in this case, the response MUST NOT be used in a reply to a subsequent request except if the cache relays the new request to the origin server in a conditional request, and the origin server responds with 304 (Not Modified) and includes the same variant-ID (see 13.8.3).

13.8.2 SLUSHY: Use of the Alternates header

Origin servers may respond to requests for varying resources with a status of 300 (Multiple choice), using the `Alternates` header (see section 10.alternates) to inform the requesting client that describes the set of possible choices, including specific URIs for each variant.

Roy says this response also includes a `Content-Location` header.

In this case, the client may choose one of the available variants and make a subsequent request using the specific URI for that variant. Since such an URI is bound to just one entity, the origin server's response to this request includes neither a `Vary` header nor an `Alternates` header, and a cache may treat it as it would any non-varying resource.

If a cache receives an `Alternates` header in a response from the origin server, it should act as if the response carried a "`Vary:{accept-headers}`" header. This means that the response may be returned in reply to a subsequent request with `Accept-*` headers identical to those in the current request.

> Note that section 13.14.1 prevents caching of 300 (Multiple choices) responses unless this is explicitly allowed by an `Expires` or `Cache-control` header.

13.8.3 Use of Variant-IDs

A cache stores copies of specific entity instances, not copies of varying resources per se. Therefore, the URI of a varying resource is not sufficient for use as a cache key. In certain interactions between a cache and an origin server, it is convenient to encode the cache key using a more compact representation than the full set of selecting request headers. Or, if the selection criteria are not known to the cache, it may be impossible to express the actual cache key to the cache. For these reasons, the HTTP protocol provides two different optional mechanisms to encode a cache key:

- Variant-IDs: an opaque identifier for a specific variant of a varying resource.
- Selecting opaque validators: a special kind of opaque validator that is defined to be unique across all variants of a varying resource.

Variant-IDs are the preferred mechanism, since they generally allow more efficient management of caches.

If an origin server chooses to use the variant-ID mechanism, it assigns a variant-ID (see section 3.14) to each distinct variant. This assignment can only be done by the origin server. It then returns the appropriate variant-ID with each response that applies to a specific variant, using the `CVal` header (see 10.47).

If an origin server provides a variant-ID for any variant of a resource, it SHOULD provide a variant-ID for all variants of that resource.

When a cache receives a successful response with a variant-ID, it SHOULD use this information to replace any existing cache entries for the same variant of the corresponding URI. That is, it forms a cache key using the URI of the request and the variant-ID of the response. If this key matches the key of an existing cache entry, it SHOULD replace the existing entry with the new response (subject to all of the other rules on caching). See section 13.12 for more details on update.

When a cache performs a conditional request on a varying resource, and it has one or more cache entries for the resource that include variant-IDs, the cache MUST transmit the (cache-validator, variant-ID) tuples in the conditional request, using the variant-set mechanism (see section 3.16). This tells the server which variants are currently in the requester’s cache.

> The client MAY choose to transmit only a subset of the (cache-validator, variant-ID) tuples corresponding to its cache entries for this resource.

When a server receives a conditional request that includes a variant-set, and the server is able to reply with an appropriate variant (either because it is the origin server, or because it is an intermediate cache that can properly implement the variant selection algorithm), once it has selected the variant it should examine the elements of the supplied variant-set. If one of these matches the variant-ID of the selected variant, and if the cache validators match, the server SHOULD reply with a 304 (Not Modified) response, including the variant-ID of the selected variant. Otherwise, the server should reply as if the request were unconditional.

The server may optionally use the variant-set information in its selection algorithm. For example, if the selection algorithm yields several variants with equal preference, and one of these is already in the requester's cache, the server could select that variant and avoid an extra data transfer. This is a performance optimization; otherwise, the variant-selection mechanism is orthogonal to the variant-ID mechanism.

13.8.4 Use of Selecting Opaque Validators

If the origin server prefers not to provide variant-IDs, it MAY at its option use the "selecting opaque validator" mechanism. A selecting opaque validator is an opaque validator whose value is unique across all variants of a resource.

If the origin server cannot generate opaque validators that are guaranteed to be unique across all variants of a varying resource, it MUST NOT send any opaque validators for that resource.

When a cache receives a successful response with an opaque validator and no variant-ID, it MAY either replace any cache entries for the resource with the new response, or it may keep multiple such entries. See section 13.12 for more details on update.

When a cache performs a conditional request on a varying resource, and it has one or more cache entries for the resource that include opaque validators, the cache SHOULD transmit the set of opaque validators in the conditional request, using the validator-set mechanism (see section 3.15). This tells the server which variants are currently in the requester's cache.

The client MAY chose to transmit only a subset of the opaque validators from its cache entries for this resource.

When a server receives a conditional request that includes a validator-set, and the server is able to reply with an appropriate variant (either because it is the origin server, or because it is an intermediate cache that can properly implement the variant selection algorithm), once it has selected the variant it should examine the elements of the supplied validator-set. If one of these matches the cache validator of the selected variant, the server SHOULD reply with a 304 (Not Modified) response, including that cache validator. Otherwise, the server should reply as if the request were unconditional.

13.10 Shared and Non-Shared Caches

For reasons of security and privacy, it is necessary to make a distinction between "shared" and "non-shared" caches. A non-shared cache is one that is accessible only to a single user. Accessibility in this case SHOULD be enforced by appropriate security mechanisms. All other caches are considered to be "shared." Other sections of this specification place certain constraints on the operation of shared caches in order to prevent loss of privacy or failure of access controls.

13.11 SLUSHY: Miscellaneous Considerations

This section is somewhat miscellaneous, and its contents might be shifted to other locations in the document.

13.11.1 Detecting Firsthand Responses

Note that a client can usually tell if a response is firsthand by comparing the `Date` to its local request-time, and hoping that the clocks are not badly skewed.

13.11.2 Disambiguating Expiration values

Because expiration values are assigned optimistically, it is possible that two caches may contain fresh values for the same resource that are different.

If a client performing a retrieval receives a non-firsthand response for a resource that was already fresh in its own cache, and the `Date` header in its existing cache entry is newer than the `Date` on the new response, then the client MAY ignore the response. If so, it MAY retry the request with a "`Cache-Control`: max-age=0" directive (see section 10.8), to force a check with the origin server.

If a cache that is pooling cached responses from other caches sees two fresh responses for the same resource with different validators, it SHOULD use the one with the newer `Date` header.

13.11.3 Disambiguating Multiple Responses

Because a client may be receiving responses via multiple paths, so that some responses flow through one set of caches and other responses flow through a different set of caches, a client may receive responses in an order different from that in which the origin server generated them. We would like the client to use the most recently generated response, even if older responses are still apparently fresh.

Neither the opaque validator nor the expiration value can impose an ordering on responses, since it is possible that a later response intentionally carries an earlier expiration time. However, the HTTP/1.1 specification requires the transmission of `Date` headers on every response, and the `Date` values are ordered to a granularity of one second.

If a client performs a request for a resource that it already has in its cache, and the response it receives contains a `Date` header that appears to be older than the one it already has in its cache, then the client SHOULD repeat the request unconditionally, and include

```
Cache-Control: max-age=0
```

to force any intermediate caches to validate their copies directly with the origin server, or

```
Cache-Control: no-cache
```

to force any intermediate caches to obtain a new copy from the origin server. This prevents certain paradoxes arising from the use of multiple caches.

If the `Date` values are equal, then the client may use either response (or may, if it is being extremely prudent, request a new response). Servers MUST NOT depend on clients being able to choose deterministically between responses generated during the same second, if their expiration times overlap.

13.12 SLUSHY: Cache Keys

A "cache key" is a value used to identify a cache entry. HTTP caches three different kinds of cache keys, for use in different contexts:

- Some subset of the fields stored with a cache entry constitute the "entry key" for that entry. These may include the `Request-URI`, some request-header fields, and some response-header fields.
- Some subset of the fields of a response, together with perhaps the `Request-URI`, constitute the "update key" of a response.
- Some subset of the fields of a request, together with the `Request-URI`, constitute the "lookup key" of a request.

When a cache receives a request, it builds a lookup key from that request, then tries to find (lookup) a cache entry with a matching entry key according to the key matching procedure in section 13.12.3. If such a match exists, then the cache can decide (based on the other caching rules) whether to return that entry in reply to the request.

When a cache receives a response, it builds a update key from that response, and from the request that elicited it. It uses this key to find any previously stored entry with a matching entry key. If such an entry exists, the cache replaces the old entry with the new one.

> The term "update" means to remove the old entry from the cache, and then to insert the new entry. It does not imply a modification of an existing entry.

This section describes specifically how the three kinds of keys are constructed, and how a cache determines if keys match.

13.12.1 Non-varying Resources

When a response is received for a non-varying resource (that is, the response includes no `Vary`, `Alternates`, or `Content-Location` headers), the update key for the response is simply the `Request-URI` of the request that elicited it: (`Request-URI`, null). The entry key for the response is (`Request-URI`, null, null).

13.12.2 SLUSHY: Varying Resources

If a response includes a `Vary` header, then we use the notation "`sel-hdr-values`" to denote the canonical form of the headers in the corresponding request whose field-names are given in the `Vary` header. If the response does not include a `Vary` header, then `sel-hdr-values` is assigned the null value. Section 10.52 on `Vary` defines the canonical form for selecting headers.

The canonical form of the headers is defined to be a set whose elements are sequences of request headers with identical field-names. For a given field-name, the corresponding element is the concatenation of the request headers with that field-name, in exactly the order that these fields appear in the request

If the response contains "`Vary: {other}`", then `sel-hdr-values` is assigned a non-null value that is defined as never matching a set of request headers.

When a response is received that includes a variant-ID in a `CVal` header (see section 10.102), but no `Content-Location` header, then the update key is (`Request-URI`, variant-ID), and the entry key for the response is (`Request-URI`, variant-ID, `sel-hdr-values`).

When a response is received that includes a `Vary` header and an opaque validator, but no variant-ID or `Content-Location` header, then the update key is (`Request-URI`, opaque-validator), and the entry key for the response is (`Request-URI`, opaque-validator, `sel-hdr-values`).

This rule supports the "selecting opaque validators" mechanism described in section 13.8.4. The cache should distinguish between actual variant-IDs and opaque-validators in the variant-ID element of the entry key; a non-null opaque-validator in an entry key DOES match a null variant-ID in a lookup key.

When a response is received that includes both a variant-ID in a `CVal` header, and a `Content-Location` header, then the update key is (`content-location-URI`, `variant-ID`), and the entry key for the response is (`content-location-URI`, `variant-ID`, `sel-hdr-values`).

When a response is received that includes a `Content-Location` header but no variant-ID, then the update key is (`content-location-URI`, null), and the entry key for the response is (content-location-URI, null, `sel-hdr-values`).

13.12.3 SLUSHY: Key-Matching Procedure

We express entry keys as the tuple (`URI`, `variant-ID`, `sel-hdr-values`), in which the variant-ID may be null, and the sel-hdr-values may either be null, or may be a set of request headers.

We express update keys as a tuple (URI, `variant-ID`), in which the `variant-ID` may be null. A update key matches an entry key if both their URI elements match and their `variant-ID` elements match. (A null `variant-ID` does not match a non-null `variant-ID`.)

We express lookup keys as a tuple (`URI`, `variant-ID`, `all-request-headers`), in which the `variant-ID` may be null. The all-request-headers element of the tuple is not always used, but is included here as a notational convenience. A lookup key matches an entry key if both their URI elements match and their variant-ID elements match, and either

- the `sel-hdr-values` element of the entry key is null

or

- the `sel-hdr-values` element of the entry key matches the appropriate headers in the all-request-headers element of the lookup key, according to the matching rules in section on `Vary`, section 10.52.

This description matching algorithm is clearly not the most efficient implementation of an equivalent algorithm. A cache may use any algorithm that yields equivalent results. For example, it may use a hierarchical approach where cache entries are grouped into sets by the URI and variant-ID, and only if a set includes non-null `sel-hdr-values` elements does the cache need to consider the other request headers.

If on a cache lookup there are two or more fresh entries that appear to match the request, then the one with the most recent `Date` value MUST be used.

13.12.4 Canonicalization of URIs

A cache, when comparing two URIs to decide if they match or not, a cache MUST use a case-sensitive octet-by-octet comparison of the entire URIs, with these exceptions:

Following the rules from section 3.2.2:

- A port that is empty or not given is equivalent to port 80.
- Comparisons of host names MUST be case-insensitive.
- Comparisons of scheme names MUST be case-insensitive.
- An empty abs_path is equivalent to an abs_path of "/"

Characters except those in the reserved set and the unsafe set (see section 3.2) are equivalent to their "`"%" HEX HEX`" encodings.

For example, the following three URIs are equivalent:

```
http://abc.com:80/~smith/home.html
http://ABC.com/%7Esmith/home.html
http://ABC.com:/%7esmith/home.html
```

13.13 FLUID: Cache-Related Problems Not Addressed in HTTP/1.1

TBS

This section will list a few problems that are NOT addressed in HTTP/1.1, with the intention of encouraging implementers not to adopt proprietary solutions inconsistent with possible future protocol revisions..

- Server-driven invalidation
- Demographics

13.14 Cache Operation When Receiving Errors or Incomplete Responses

A cache that receives an incomplete response (for example, with fewer bytes of data than specified in a Content-length: header) may store the response. However, the cache MUST treat this as a partial response. Partial responses may be combined as described in section 13.7.4; the result might be a full response or might still be partial. A cache MUST NOT return a partial response to a client without explicitly marking it as such, using the 206 (Partial Content) status code. A cache MUST NOT return a partial response using a status code of 200 (OK).

A cache that receives a response with a zero-length Entity-body and no explicit indication that the correct length is zero (such as "`Content-Length: 0`") MUST NOT not store the response. The same rule applies to a response of any length received without an explicit length indication if the transport connection was terminated in any unusual way.

If a cache receives a response carrying `Retry-After` header (see section 10.36), it may either forward this response to the requesting client, or act as if the server failed to respond. In the latter case, it MAY return a previously received response, although it MUST follow all of the rules applying to stale responses. In particular, it MUST NOT override the "`must-revalidate`" `Cache-Control` directive (see section 10.7).

13.14.1 Caching and Status Codes

A response received with a status code of 200 or 206 may be stored by a cache and used in reply to a subsequent request, subject to the expiration mechanism, unless a Cache-control directive prohibits caching.

A response received with any other status code MUST not be returned in a reply to a subsequent request unless it carries at least one of the following:

- an `Expires` header
- a max-age `Cache-control` directive
- a must-revalidate `Cache-control` directive
- a public `Cache-control` directive

13.14.2 Handling of Retry-After

If a cache receives a response carrying a Retry-After header (see section 10.36), it may either forward this response to the requesting client, or act as if the server failed to respond. In the latter case, it MAY return a previously received response, although it MUST follow all of the rules applying to stale responses. In particular, it MUST not override the "`must-revalidate`" `Cache-control` directive (see section 10.7).

13.15 FLUID: Compatibility With Earlier Versions of HTTP

TBS

If anything should be here, it should be a collection of warnings about what HTTP/1.1 systems should not assume about HTTP/1.0 systems.

13.16 SLUSHY: Side Effects of GET and HEAD

Unless the origin server explicitly prohibits the caching of their responses, the application of GET and HEAD methods to any resources SHOULD NOT have side effects that would lead to erroneous behavior if these responses are taken from a cache. They may still have side effects, but a cache is not required to consider such side effects in its caching decisions. Caches are always expected to observe an origin server's explicit restrictions on caching.

We note one exception to this rule: since some applications have traditionally used GETs and HEADs with query URLs (those containing a “?” in the rel_path part) to perform operations with significant side effects, caches MUST NOT treat responses to such URLs as fresh unless the server provides an explicit expiration time.

This specifically means that responses from HTTP/1.0 servers for such URIs should not be taken from a cache.

See section 15.2 for related information.

13.17 SLUSHY: Invalidation After Updates or Deletions

The effect of certain methods at the origin server may cause one or more existing cache entries to become non-transparently invalid. That is, although they may continue to be “fresh,” they do not accurately reflect what the origin server would return for a new request.

There is no way for the HTTP protocol to guarantee that all such cache entries are marked invalid. For example, the request that caused the change at the origin server may not have gone through the proxy where a cache entry is stored. However, several rules help reduce the likelihood of erroneous behavior.

In this section, the phrase “invalidate an entity” means that the cache should either remove all instances of that entity from its storage, or should mark these as “invalid” and in need of a mandatory revalidation before they can be returned in response to a subsequent request.

Some HTTP methods invalidate a single entity. This is either the entity referred to by the `Request-URI`, or by the `Location` or `Content-Location` response headers (if present). These methods are:

- PUT
- DELETE
- POST

In order to prevent denial of service attacks, an invalidation based on the URI in a `Location` or `Content`-Location header MUST only be performed if the host part is the same as in the Request-`URI`.

13.18 Write-Through Mandatory

All methods that may be expected to cause modifications to the origin server's resources MUST be written through to the origin server. This currently includes all methods except for GET and HEAD. A cache MUST NOT reply to such a request from a client before having transmitted the request to the inbound server, and having received a corresponding response from the inbound server.

The alternative (known as “write-back” or “copy-back” caching) is not allowed in HTTP/1.1, due to the difficulty of providing consistent updates and the problems arising from server, cache, or network failure prior to write-back.

13.19 Interoperability of Varying Resources with HTTP/1.0 Proxy Caches

If the correct handling of responses from a varying resource (Section 10.xxx) by HTTP/1.0 proxy caches in the response chain is important, HTTP/1.1 origin servers can include the following `Expires` (Section 10.exp) response header in all responses from the varying resource:

```
Expires: Thu, 01 Jan 1980 00:00:00 GMT
```

If this `Expires` header is included, the server should usually also include a `Cache-Control` header for the benefit of HTTP/1.1 caches, for example

```
Cache-Control: max-age=604800
```

which overrides the freshness lifetime of zero seconds specified by the included `Expires` header.

13.20 Cache Replacement for Varying Resources

If a new 200 (OK) response is received from a non-varying resource while an old 200 (OK) response is cached, caches can delete this old response from cache memory and insert the new response. For 200 (OK) responses from varying resources (Section 13.12.3), cache replacement is more complex.

HTTP/1.1 allows the authors of varying resources to guide cache update by the inclusion of elements of so-called update keys in the responses of these resources. The update key of a varying response consists of two elements, both of which may be empty strings, separated by a semicolon:

```
        update-key  =  variant-id ";" absoluteURI
```

The variant-id element of the update key is the `variant-id` value in the `CVal` header of the response, if a `CVal` header which such a value is present, and an empty string otherwise. The absoluteURI element of the update key is the absolute URI given in, or derived from, the `Content-Location` header of the response if present, and an empty string if no `Content-Location` header is present.

If a cache has stored in memory a 200 (OK) response with a certain update key, and receives, from the same resource, a new 200 (OK) response which has the same update key, this should be interpreted as a signal from the resource author that the old response can be deleted from cache memory and replaced by the new response.

The update key mechanism cannot cause deletion from cache memory of old responses with update keys that will no longer be used. It is expected that the normal "least recently used" update heuristics employed by caches will eventually cause such old responses to be deleted.

All 200 (OK) responses from varying resources should include update key elements. Resource authors may not assume that caches will be able to cache responses not including update key elements. If a Vary header is used to signal variance, the response should include a `variant-id` value as the update key element. The `Content-Location` header should only be used to supply a update key element if an `Alternates` header is present in the response.

13.22 FLUID: Network Partitions

TBS

There may be enough said elsewhere already, but we haven't checked.

13.23 FLUID: Caching of Negative Responses

TBS

13.24 History Lists

History lists as implemented in many user agents and caches are different. In particular history lists SHOULD NOT try to show a semantically transparent view of the current state of a resource. Rather, a history list is meant to show exactly what the user saw at the time when the resource was retrieved .

This should not be construed to prohibit the history mechanism from telling the user that a view may be stale.

14 Persistent Connections

14.1 Purpose

HTTP's greatest strength and its greatest weakness has been its simplicity. Prior to persistent connections, a separate TCP connection was established to fetch each URL, increasing the load on HTTP servers, and causing congestion on the Internet. The use of inline images and other associated data often requires a client to make multiple requests of the same server in a short amount of time. An excellent analysis of these performance problems is available [2]; analysis and results from a prototype implementation are in [32, 33].

Persistent HTTP connections have a number of advantages, including:

- By opening and closing TCP fewer connections, CPU time is saved, and memory used for TCP protocol control blocks is also saved
- HTTP requests and responses can be pipe-lined on a connection. Pipe-lining allows a client to make multiple requests without waiting for each response, allowing a single TCP connection to be used much more efficiently, with much lower elapsed time.
- Network congestion is reduced by reducing the number of packets caused by TCP opens, and by allowing TCP sufficient time to determine the congestion state of the network.
- HTTP can evolve more gracefully; since errors can be reported without the penalty of closing the TCP connection. Clients using future versions of HTTP might optimistically try a new feature, but if communicating with an older server, retry with old semantics after an error were reported.

HTTP implementations SHOULD implement persistent connections.

14.2 Overall Operation

Persistent connections provides a mechanism by which a client and a server can negotiate the use of a TCP connection for an extended conversation.. This negotiation takes place using the `Connection` and `Persist` header fields. Once this option has been negotiated the client can make multiple HTTP requests over a single transport connection.

14.2.3 Negotiation

To request the use of persistent connections, a client sends a `Connection` header with a connection-token "`Persist`". If the server wishes to accept persistent connections it will respond with the same connection-token. Both the client and server MUST send this connection-token with every request and response for the duration of the persistent connection. If either the client or the server omits the `Persist` token from the `Connection` header, that request becomes the last one for the connection.

A server MUST NOT establish a persistent connection with an HTTP/1.0 client that uses the above form of the `Persist` header due to problems with the interactions between 1.1 clients and 1.0 proxy servers (See section E.2.5 for more information on backwards compatibility with HTTP 1.0 clients).

14.2.4 Pipe-lining

Clients and servers which support persistent connections MAY "pipe-line" their requests and responses. When pipe-lining, a client will send multiple requests without waiting for the responses. The server MUST then send all of the responses in the same order that the requests were made.

A client MAY pipeline multiple requests immediately if it has previous knowledge that the server it is connecting to supports persistent connections. A client MAY assume that a server supports persistent connections if the same server has accepted persistent connections within the past 24 hours. Clients which assume persistent connections and pipeline immediately SHOULD be prepared to retry their connection if the first pipe-lined attempt fails. If a client does such a retry, it MUST NOT pipeline without first receiving an explicit `Persist` token from the server. Clients MUST also be prepared to resend their requests if the server closes the connection before sending all of the corresponding responses.

14.2.5 Delimiting Entity-Bodies

When using persistent connections both the client and the server MUST mark the exact endings of transmitted entity-bodies using one of the following three techniques:

1. Send a `Content-length` field in the header with the exact number of bytes in the entity-body.
2. Send the message using Chunked transfer encoding as described in section 3.6. Chunked transfer encoding allows the server to transmit the data to the client a piece at a time while still communicating an exact ending of the entity-body.
3. Close the transport connection after the entity body.

Sending the `Content-length` is the preferred technique. Chunked encoding SHOULD be used when the size of the entity-body is not known before beginning to transmit the entity-body. Finally, the connection MAY be closed and fall back to non-persistent connections, if neither 1 or 2 are possible.

Clients and servers that support persistent connections MUST correctly support receiving via all three techniques.

14.3 Proxy Servers

It is especially important that proxies correctly implement the properties of the `Connection` header field as specified in 14.2.1.

The proxy server MUST negotiate persistent connections separately with its clients and the origin servers (or other proxy servers) that it connects to. Each persistent connection applies to only one transport link.

A proxy server MUST NOT establish a persistent connection with an HTTP 1.0 client.

14.4 Interaction with Security Protocols

It is expected that the Session extension will operate with both SHTTP [31] and SSL [32]. When used in conjunction with SHTTP, the SHTTP request is prepared normally and the persist connection-token is placed in the outermost request block (the one containing the “Secure” method). When used in conjunction with SSL, a SSL session is started as normal and the first HTTP request made using SSL contains the persistent connection header.

14.5 Practical Considerations

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 server. The use of persistent connections places no requirements on the length of this time-out for either the client or the server.

When a client or server wishes to time-out it SHOULD issue a graceful close on the transport connection. Clients and servers SHOULD both constantly watch for the other side of the transport close, and respond to it as appropriate. If a client or server does not detect the other sides close promptly it could cause unnecessary resource drain on the network.

A client, server, or proxy MAY close the transport connection at any time. For example, a client MAY 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.

This means that clients, servers, and proxies MUST be able to recover from asynchronous close events. Client software SHOULD reopen the transport connection and retransmit the aborted request without user interaction. However, this automatic retry SHOULD NOT be repeated if the second request fails.

Servers SHOULD always respond to at least one request per connection, if at all possible. Servers SHOULD NOT close a connection in the middle of transmitting a response, unless a network or client failure is suspected.

It is suggested that clients which use persistent connections SHOULD limit the number of simultaneous connections that they maintain to a given server. A single-user client SHOULD maintain AT MOST 2 connections with any server of proxy. A proxy SHOULD use up to 2*N connections to another server or proxy, where N is the number of simultaneously active users. These guidelines are intended to improve HTTP response times and avoid congestion of the Internet or other networks.

15. Security Considerations

This section is meant to inform application developers, information providers, and users of the security limitations in HTTP/1.1 as described by this document. The discussion does not include definitive solutions to the problems revealed, though it does make some suggestions for reducing security risks.

15.1 Authentication of Clients

As mentioned in Section 11.1, the Basic authentication scheme is not a secure method of user authentication, nor does it in any way protect the `Entity-Body`, which is transmitted in clear text across the physical network used as the carrier. HTTP does not prevent additional authentication schemes and encryption mechanisms from being employed to increase security or the addition of enhancements (such as schemes to use one-time passwords) to Basic authentication.

The most serious flaw in Basic authentication is that it results in the essentially clear text transmission of the user's password over the physical network. It is this problem which Digest Authentication attempts to address.

Because Basic authentication involves the clear text transmission of passwords it SHOULD never be used (without enhancements) to protect sensitive or valuable information.

A common use of Basic authentication is for identification purposes -- requiring the user to provide a user name and password as a means of identification, for example, for purposes of gathering accurate usage statistics on a server. When used in this way it is tempting to think that there is no danger in its use if illicit access to the protected documents is not a major concern. This is only correct if the server issues both user name and password to the users and in particular does not allow the user to choose his or her own password. The danger arises because naive users frequently reuse a single password to avoid the task of maintaining multiple passwords.

If a server permits users to select their own passwords, then the threat is not only illicit access to documents on the server but also illicit access to the accounts of all users who have chosen to use their account password. If users are allowed to choose their own password that also means the server must maintain files containing the (presumably encrypted) passwords. Many of these may be the account passwords of users perhaps at distant sites. The owner or administrator of such a system could conceivably incur liability if this information is not maintained in a secure fashion.

Basic Authentication is also vulnerable to spoofing by counterfeit servers. If a user can be led to believe that he is connecting to a host containing information protected by basic authentication when in fact he is connecting to a hostile server or gateway then the attacker can request a password, store it for later use, and feign an error. This type of attack is not possible with Digest Authentication[26]. Server implementers SHOULD guard against the possibility of this sort of counterfeiting by gateways or CGI scripts. In particular it is very dangerous for a server to simply turn over a connection to a gateway since that gateway can then use the persistent connection mechanism to engage in multiple transactions with the client while impersonating the original server in a way that is not detectable by the client.

15.2 Safe Methods

The writers of client software should be aware that the software represents the user in their interactions over the Internet, and should be careful to allow the user to be aware of any actions they may take which may have an unexpected significance to themselves or others.

In particular, the convention has been established that the `GET` and `HEAD` methods should never have the significance of taking an action other than retrieval. These methods should be considered "safe. " This allows user agents to represent other methods, such as `POST`, `PUT` and `DELETE`, in a special way, so that the user is made aware of the fact that a possibly unsafe action is being requested.

Naturally, it is not possible to ensure that the server does not generate side-effects as a result of performing a `GET` request; in fact, some dynamic resources consider that a feature. The important distinction here is that the user did not request the side-effects, so therefore cannot be held accountable for them.

15.3 Abuse of Server Log Information

A server is in the position to save personal data about a user's requests which may identify their reading patterns or subjects of interest. This information is clearly confidential in nature and its handling may be constrained by law in certain countries. People using the HTTP protocol to provide data are responsible for ensuring that such material is not distributed without the permission of any individuals that are identifiable by the published results.

15.4 Transfer of Sensitive Information

Like any generic data transfer protocol, HTTP cannot regulate the content of the data that is transferred, nor is there any a priori method of determining the sensitivity of any particular piece of information within the context of any given request. Therefore, applications SHOULD supply as much control over this information as possible to the provider of that information. Four header fields are worth special mention in this context: `Server`, `Via`, `Referer` and `From`.

Revealing the specific software version of the server may allow the server machine to become more vulnerable to attacks against software that is known to contain security holes. Implementers SHOULD make the `Server` header field a configurable option.

Proxies which serve as a portal through a network firewall SHOULD take special precautions regarding the transfer of header information that identifies the hosts behind the firewall. In particular, they SHOULD remove, or replace with sanitized versions, any `Via` fields generated behind the firewall.

The `Referer` field allows reading patterns to be studied and reverse links drawn. Although it can be very useful, its power can be abused if user details are not separated from the information contained in the `Referer`. Even when the personal information has been removed, the `Referer` field may indicate a private document's URI whose publication would be inappropriate.

The information sent in the From field might conflict with the user's privacy interests or their site's security policy, and hence it SHOULD not be transmitted without the user being able to disable, enable, and modify the contents of the field. The user MUST be able to set the contents of this field within a user preference or application defaults configuration.

We suggest, though do not require, that a convenient toggle interface be provided for the user to enable or disable the sending of `From` and `Referer` information.

15.5 Attacks Based On File and Path Names

Implementations of HTTP origin servers SHOULD be careful to restrict the documents returned by HTTP requests to be only those that were intended by the server administrators. If an HTTP server translates HTTP URIs directly into file system calls, the server MUST take special care not to serve files that were not intended to be delivered to HTTP clients. For example, UNIX, Microsoft Windows, and other operating systems use ".." as a path component to indicate a directory level above the current one. On such a system, an HTTP server MUST disallow any such construct in the `Request-URI` if it would otherwise allow access to a resource outside those intended to be accessible via the HTTP server. Similarly, files intended for reference only internally to the server (such as access control files, configuration files, and script code) MUST be protected from inappropriate retrieval, since they might contain sensitive information. Experience has shown that minor bugs in such HTTP server implementations have turned into security risks.

15.6 Personal Information

HTTP clients are often privy to large amounts of personal information (e.g. the user's name, location, mail address, passwords, encryption keys, etc.), and SHOULD be very careful to prevent unintentional leakage of this information via the HTTP protocol to other sources. We very strongly recommend that a convenient interface be provided for the user to control dissemination of such information, and that designers and implementers be particularly careful in this area. History shows that errors in this area are often both serious security and/or privacy problems, and often generate very adverse publicity for the implementer's company.

15.7 Privacy issues connected to Accept headers

`Accept` request headers can reveal information about the user to all servers which are accessed. The `Accept-Language` header in particular can reveal information the user would consider to be of a private nature, because the understanding of particular languages is often strongly correlated to the membership of a particular ethnic group. User agents which offer the option to configure the contents of an `Accept-Language` header to be sent in every request are strongly encouraged to let the configuration process include a message which makes the user aware of the loss of privacy involved.

An approach that limits the loss of privacy would be for a user agent to omit the sending of Accept-Language headers by default, and to ask the user whether it should start sending Accept-Language headers to a server if it detects, by looking for any Vary or Alternates response headers generated by the server, that such sending could improve the quality of service.

Elaborate user-customized accept header fields sent in every request, in particular if these include quality values, can be used by servers as relatively reliable and long-lived user identifiers. Such user identifiers would allow content providers to do click-trail tracking, and would allow collaborating content providers to match cross-server click-trails or form submissions of individual users. Note that for many users not behind a proxy, the network address of the host running the user agent will also serve as a long-lived user identifier. In environments where proxies are used to enhance privacy, user agents should be conservative in offering accept header configuration options to end users. As an extreme privacy measure, proxies could filter the accept headers in relayed requests. General purpose user agents which provide a high degree of header configurability should warn users about the loss of privacy which can be involved.

15.8 DNS Spoofing

Clients using HTTP rely heavily on the Domain Name Service, and are thus generally prone to security attacks based on the deliberate miss-association of IP addresses and DNS names. The deployment of DNSSEC[27] should help this situation. In advance of this deployment, however, clients need to be cautious in assuming the continuing validity of an IP number/DNS name association.

In particular, HTTP clients SHOULD rely on their name resolver for confirmation of an IP number/DNS name association, rather than caching the result of previous host name lookups. Many platforms already can cache host name lookups locally when appropriate, and they SHOULD be configured to do so. These lookups should be cached, however, only when the TTL (Time To Live) information reported by the name server makes it likely that the cached information will remain useful.

If HTTP clients cache the results of a host name lookups in order to achieve a performance improvement, they MUST observe the TTL information reported by DNS.

If HTTP clients do not observe this rule, they could be spoofed when a previously-accessed server's IP address changes. As renumbering is expected to become increasingly common[24], the possibility of this form of attack will grow. Observing this requirement thus reduces this potential security vulnerability.

This requirement also improves the load-balancing behavior of clients for replicated servers using the same DNS name and reduces the likelihood of a user's experiencing failure in accessing sites which use that strategy.

15.9 SLUSHY: Location Headers and Spoofing

If a single server supports multiple organizations that do not trust one another, then it must check the values of `Location` and `Content-Location` headers in responses that are generated under control of said organizations to make sure that they do not attempt to invalidate resources over which they have no authority.

16. Acknowledgments

This specification makes heavy use of the augmented BNF and generic constructs defined by David H. Crocker for RFC 822 [9]. Similarly, it reuses many of the definitions provided by Nathaniel Borenstein and Ned Freed for MIME [7]. We hope that their inclusion in this specification will help reduce past confusion over the relationship between HTTP and Internet mail message formats.

The HTTP protocol has evolved considerably over the past four years. It has benefited from a large and active developer community--the many people who have participated on the *www-talk* mailing list--and it is that community which has been most responsible for the success of HTTP and of the World-Wide Web in general. Marc Andreessen, Robert Cailliau, Daniel W. Connolly, Bob Denny, John Franks, Jean-Francois Groff, Phillip M. Hallam-Baker, Håkon W. Lie, Ari Luotonen, Rob McCool, Lou Montulli, Dave Raggett, Tony Sanders, and Marc VanHeyningen deserve special recognition for their efforts in defining early aspects of the protocol.

This document has benefited greatly from the comments of all those participating in the HTTP-WG. In addition to those already mentioned, the following individuals have contributed to this specification:

Gary Adams
Harald Tveit Alvestrand
Keith Ball
Brian Behlendorf
Paul Burchard
Maurizio Codogno
Mike Cowlishaw
Roman Czyborra
Michael A. Dolan
Jim Gettys
Marc Hedlund
Koen Holtman
Alex Hopmann
Bob Jernigan
Shel Kaphan
Rohit Khare
Martijn Koster
Alexei Kosut
David M. Kristol
Daniel LaLiberte
Paul J. Leach
Albert Lunde
John C. Mallery
Jean-Philippe Martin-Flatin
Larry Masinter
Mitra
Jeffrey Mogul
Gavin Nicol
Bill Perry
Jeffrey Perry
Owen Rees
Luigi Rizzo
David Robinson
Marc Salomon
Rich Salz
Jim Seidman
Chuck Shotton
Eric W. Sink
Simon E. Spero
Richard N. Taylor
Robert S. Thau
François Yergeau
Mary Ellen Zurko
David Morris
Greg Herlihy
Scott Powers
Allan M. Schiffman
Alan Freier
Bill (BearHeart) Weinman

Much of the content and presentation of the caching design is due to suggestions and comments from individuals including: Shel Kaphan, Paul Leach, Koen Holtman, David Morris, Larry Masinter, and Roy Fielding.

Most of the specification of ranges is based on work originally done by Ari Luotonen and John Franks, with additional input from Steve Zilles and Roy Fielding.

XXX need acks for subgroup work.

17. References

[1]
H. Alvestrand. "Tags for the identification of languages." RFC 1766, UNINETT, March 1995.

[2]
F. Anklesaria, M. McCahill, P. Lindner, D. Johnson, D. Torrey, B. Alberti. "The Internet Gopher Protocol: (a distributed document search and retrieval protocol)", RFC 1436, University of Minnesota, March 1993.

[3]
T. Berners-Lee. "Universal Resource Identifiers in WWW A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World-Wide Web." RFC 1630, CERN, June 1994.

[4]
T. Berners-Lee, L. Masinter, M. McCahill. "Uniform Resource Locators (URL)." RFC 1738, CERN, Xerox PARC, University of Minnesota, December 1994.

[5]
T. Berners-Lee, D. Connolly. "HyperText Markup Language Specification - 2.0." RFC 1866, MIT/LCS, November 1995.

[6]
T. Berners-Lee, R. Fielding, H. Frystyk. "Hypertext Transfer Protocol - HTTP/1.0." Work in Progress (draft-ietf-http-v10-spec-04.txt), MIT/LCS, UC Irvine, September 1995.

[7]
N. Borenstein, N. Freed. "MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies." RFC 1521, Bellcore, Innosoft, September 1993.

[8]
R. Braden. "Requirements for Internet hosts - application and support." STD 3, RFC 1123, IETF, October 1989.

[9]
D. H. Crocker. "Standard for the Format of ARPA Internet Text Messages." STD 11, RFC 822, UDEL, August 1982.

[10]
F. Davis, B. Kahle, H. Morris, J. Salem, T. Shen, R. Wang, J. Sui, M. Grinbaum. "WAIS Interface Protocol Prototype Functional Specification." (v1.5), Thinking Machines Corporation, April 1990.

[11]
R. Fielding. "Relative Uniform Resource Locators." RFC 1808, UC Irvine, June 1995.

[12]
M. Horton, R. Adams. "Standard for interchange of USENET messages." RFC 1036 (Obsoletes RFC 850), AT&T Bell Laboratories, Center for Seismic Studies, December 1987.

[13]
B. Kantor, P. Lapsley. "Network News Transfer Protocol A Proposed Standard for the Stream-Based Transmission of News." RFC 977, UC San Diego, UC Berkeley, February 1986.

[14]
K. Moore. "MIME (Multipurpose Internet Mail Extensions) Part Two : Message Header Extensions for Non-ASCII Text." RFC 1522, University of Tennessee, September 1993.

[15]
E. Nebel, L. Masinter. "Form-based File Upload in HTML." RFC 1867, Xerox Corporation, November 1995.

[16]
J. Postel. "Simple Mail Transfer Protocol." STD 10, RFC 821, USC/ISI, August 1982.

[17]
J. Postel. "Media Type Registration Procedure." RFC 1590, USC/ISI, March 1994.

[18]
J. Postel, J. K. Reynolds. "File Transfer Protocol (FTP)" STD 9, RFC 959, USC/ISI, October 1985.

[19]
J. Reynolds, J. Postel. "Assigned Numbers." STD 2, RFC 1700, USC/ISI, October 1994.

[20]
K. Sollins, L. Masinter. "Functional Requirements for Uniform Resource Names." RFC 1737, MIT/LCS, Xerox Corporation, December 1994.

[21]
US-ASCII. Coded Character Set - 7-Bit American Standard Code for Information Interchange. Standard ANSI X3.4-1986, ANSI, 1986.

[22]
ISO-8859. International Standard -- Information Processing --
8-bit Single-Byte Coded Graphic Character Sets --
Part 1: Latin alphabet No. 1, ISO 8859-1:1987.
Part 2: Latin alphabet No. 2, ISO 8859-2, 1987.
Part 3: Latin alphabet No. 3, ISO 8859-3, 1988.
Part 4: Latin alphabet No. 4, ISO 8859-4, 1988.
Part 5: Latin/Cyrillic alphabet, ISO 8859-5, 1988.
Part 6: Latin/Arabic alphabet, ISO 8859-6, 1987.
Part 7: Latin/Greek alphabet, ISO 8859-7, 1987.
Part 8: Latin/Hebrew alphabet, ISO 8859-8, 1988.
Part 9: Latin alphabet No. 5, ISO 8859-9, 1990.

[23]
Meyers, M. Rose "The Content-MD5 Header Field." RFC 1864, Carnegie Mellon, Dover Beach Consulting, October, 1995.

[24]
B. Carpenter, Y. Rekhter, "Renumbering Needs Work". RFC 1900, IAB, February 1996.

[25]
Gzip is available from the GNU project at <URL:ftp://prep.ai.mit.edu/pub/gnu/>. A more formal specification is currently a work in progress.

[26]
Work In Progress for Digest authentication of the IETF HTTP working group.

[27]
TBS, Work in progress (XXX should put RFC in here…)

[28]
Mills, D, "Network Time Protocol, Version 3", Specification, Implementation and Analysis RFC 1305, University of Delaware, March, 1992.

[29]
Work in progress of the HTTP working group (XXX is this correct reference for incomplete work?).

[30]
S. Spero. "Analysis of HTTP Performance Problems" <URL:http://sunsite.unc.edu/mdma-release/http-prob.html>

[31]
E. Rescorla, A. Schiffman “The Secure HyperText Transfer Protocol” Internet-Draft (work in progress).

[32]
A. Freier, P Karlton, P. Kocher. “SSL Version 3.0" Internet-Draft” (work in progress).

[33]
Jeffrey C. Mogul. “The Case for Persistent-Connection HTTP”. In Proc.SIGCOMM '95 Symposium on Communications Architectures and Protocols, pages 299-313. Cambridge, MA, August, 1995.

[34]
Jeffrey C. Mogul. “The Case for Persistent-Connection HTTP”. Research, Report 95/4, Digital Equipment Corporation Western Research Laboratory, May, 1995., <URL:http://www.research.digital.com/wrl/techreports/abstracts/95.4.html>

[35]
Work in progress of the HTTP working group on state management.

18. Authors' Addresses

Roy T. Fielding
Department of Information and Computer Science
University of California
Irvine, CA 92717-3425, USA
Fax: +1 (714) 824-4056
Email: fielding@ics.uci.edu

Henrik Frystyk Nielsen
W3 Consortium
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139, USA
Fax: +1 (617) 258 8682
Email: frystyk@w3.org

Tim Berners-Lee
Director, W3 Consortium
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139, USA
Fax: +1 (617) 258 8682
Email: timbl@w3.org

Jim Gettys
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139, USA
Fax: +1 (617) 258 8682
Email: jg@w3.org

Jeffrey C. Mogul
Western Research Laboratory
Digital Equipment Corporation
250 University Avenue
Palo Alto, California, 94305, U.S.A.
Email: mogul@wrl.dec.com

Appendices

These appendices are provided for informational reasons only -- they do not form a part of the HTTP/1.1 specification.

A. Internet Media Type message/http

In addition to defining the HTTP/1.1 protocol, this document serves as the specification for the Internet media type "message/http". The following is to be registered with IANA [17].

```
Media Type name:         message

Media subtype name:      http

Required parameters:     none

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: none
```

B. Tolerant Applications

Although this document specifies the requirements for the generation of HTTP/1.1 messages, not all applications will be correct in their implementation. We therefore recommend that operational applications be tolerant of deviations whenever those deviations can be interpreted unambiguously.

Clients SHOULD be tolerant in parsing the `Status-Line` and servers tolerant when parsing the `Request-Line`. In particular, they SHOULD accept any amount of `SP` or `HT` characters between fields, even though only a single `SP` is required.

The line terminator for `HTTP-header` fields is the sequence `CRLF`. However, we recommend that applications, when parsing such headers, recognize a single `LF` as a line terminator and ignore the leading `CR`.

C. Differences Between HTTP Bodies and RFC 1521 Internet Message Bodies

HTTP/1.1 uses many of the constructs defined for Internet Mail (RFC 822 [9]) and the Multipurpose Internet Mail Extensions (MIME [7]) to allow entities to be transmitted in an open variety of representations and with extensible mechanisms. However, RFC 1521 discusses mail, and HTTP has a few features that are different than those described in RFC 1521. 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.

At the time of this writing, it is expected that RFC 1521 will be revised. The revisions may include some of the practices found in HTTP/1.1 but not in RFC 1521.

This appendix describes specific areas where HTTP differs from RFC 1521. Proxies and gateways to strict MIME environments SHOULD be aware of these differences and provide the appropriate conversions where necessary. Proxies and gateways from MIME environments to HTTP also need to be aware of the differences because some conversions may be required.

C.1 Conversion to Canonical Form

RFC 1521 requires that an Internet mail entity be converted to canonical form prior to being transferred, as described in Appendix G of RFC 1521 [7]. Section 3.6.1 of this document describes the forms allowed for subtypes of the "text" media type when transmitted over HTTP. RFC 1521 requires that content with a typeof "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 when a message is transmitted over HTTP.

Where it is possible, a proxy or gateway from HTTP to a strict RFC 1521 environment SHOULD translate all line breaks within the text media types described in Section 3.6.1 of this document to the RFC 1521 canonical form of `CRLF`. Note, however, that this may be complicated by the presence of a `Content-Encoding` and by the fact that HTTP allows the use of some character sets which do not use octets 13 and 10 to represent `CR` and `LF`, as is the case for some multi-byte character sets.

C.2 Conversion of Date Formats

HTTP/1.1 uses a restricted set of date formats (Section 3.3) to simplify the process of date comparison. Proxies and gateways from other protocols SHOULD 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.

C.3 Introduction of Content-Encoding

RFC 1521 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 MUST either change the value of the `Content-Type` header field or decode the `Entity-Body` 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 an equivalent function as `Content-Encoding`. However, this parameter is not part of RFC 1521.)

C.4 No Content-Transfer-Encoding

HTTP does not use the Content-Transfer-Encoding (CTE) field of RFC 1521. Proxies and gateways from MIME-compliant protocols to HTTP MUST remove any non-identity CTE ("quoted-printable" or "base64") 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 SHOULD label the data with an appropriate Content-Transfer-Encoding if doing so will improve the likelihood of safe transport over the destination protocol.

C.5 HTTP Header Fields in Multipart Body-Parts

In RFC 1521, most header fields in multipart body-parts are generally ignored unless the field name begins with "Content-". In HTTP/1.1, multipart body-parts may contain any HTTP header fields which are significant to the meaning of that part.

C.6 Introduction of Transfer-Encoding

HTTP/1.1 introduces the `Transfer-Encoding` header field (Section 10.39). Proxies/gateways MUST remove any transfer coding prior to forwarding a message via a MIME-compliant protocol. The process for decoding the "chunked" transfer coding (Section 3.6) can be represented in pseudo-code as:

```
length := 0
read chunk-size and CRLF
while (chunk-size > 0) {
   read chunk-data and CRLF
   append chunk-data to Entity-Body
   length := length + chunk-size
   read chunk-size and CRLF
}
read entity-header
while (entity-header not empty) {
   append entity-header to existing header fields
   read entity-header
}
Content-Length := length
Remove "chunked" from Transfer-Encoding
```

C.7 MIME-Version

HTTP is not a MIME-compliant protocol (see Appendix C). However, HTTP/1.1 messages may include a single `MIME-Version` general-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 compliance with the MIME protocol (as defined in [7]). Proxies/gateways are responsible for ensuring full compliance (where possible) when exporting HTTP messages to strict MIME environments.

```
MIME-Version   = "MIME-Version" ":" 1*DIGIT "." 1*DIGIT
```

MIME version "`1.0`" is the default for use in HTTP/1.1. However, HTTP/1.1 message parsing and semantics are defined by this document and not the MIME specification.

D. Changes from HTTP/1.0

This section will summarize major differences between versions 1.0 and 1.1 of the Hypertext Transfer Protocol.

D.1 Changes to Simplify Multi-homed Web Servers and Conserve IP Addresses

The requirements that clients and servers support the `Host` request-header, report an error if the `Host` request-header is missing from an HTTP/1.1 request (Section 10.22), and accept absolute URIs (Section 5.1.2) are among the most important changes from HTTP/1.0.

In HTTP/1.0 there is a one-to-one relationship of IP addresses and servers. There is no other way to distinguish the intended server of a request than the IP address to which that request is directed. The HTTP/1.1 change will allow the Internet, once HTTP/1.0 clients and servers are no longer common, to support multiple Web sites from a single IP address, greatly simplifying large operational Web servers, where allocation of many IP addresses to a single host has created serious problems. The Internet will also be able to recover the IP addresses that have been used for the sole purpose of allowing root-level domain names to be used in HTTP URLs. Given the rate of growth of the Web, and the number of servers already deployed, it is extremely important that implementations of HTTP/1.1 correctly implement these new requirements:

- both clients and servers MUST support the `Host` request-header
- `Host` request-headers are required in HTTP/1.1 requests.
- servers MUST report an error if an HTTP/1.1 request does not include a `Host` request-header
- servers MUST accept absolute URIs

E. Additional Features

This appendix documents protocol elements used by some existing HTTP implementations, but not consistently and correctly across most HTTP/1.1 applications. Implementers should be aware of these features, but cannot rely upon their presence in, or interoperability with, other HTTP/1.1 applications.

E.1 Additional Request Methods

E.1.1 PATCH

The `PATCH` method is similar to `PUT` except that the entity contains a list of differences between the original version of the resource identified by the `Request-URI` and the desired content of the resource after the `PATCH` action has been applied. The list of differences is in a format defined by the media type of the entity (e.g., “application/diff”) and MUST include sufficient information to allow the server to recreate the changes necessary to convert the original version of the resource to the desired version.

If the request passes through a cache and the `Request-URI` identifies a currently cached entity, that entity MUST be removed from the cache. Responses to this method are not cachable.

For compatibility with HTTP/1.0 applications, all `PATCH` requests MUST include a valid `Content-Length` header field unless the server is known to be HTTP/1.1 compliant. When sending a `PATCH` request to an HTTP/1.1 server, a client MUST use a valid `Content-Length` or the “chunked” `Transfer-Encoding`. The server SHOULD respond with a 400 (bad request) message if it cannot determine the length of the request message's content, or with 411 (length required) if it wishes to insist on receiving a valid `Content-Length`.

The actual method for determining how the patched resource is placed, and what happens to its predecessor, is defined entirely by the origin server. If the original version of the resource being patched included a `Content-Version` header field, the request entity MUST include a `Derived-From` header field corresponding to the value of the original `Content-Version` header field. Applications are encouraged to use these fields for constructing versioning relationships and resolving version conflicts.

PATCH requests must obey the entity transmission requirements set out in section 8.4.1.

Caches that implement PATCH should invalidate cached responses as defined in section 13.17 for PUT.

E.1.2 LINK

The LINK method establishes one or more Link relationships between the existing resource identified by the `Request-URI` and other existing resources. The difference between LINK and other methods allowing links to be established between resources is that the LINK method does not allow any `Entity-Body` to be sent in the request and does not directly result in the creation of new resources.

If the request passes through a cache and the `Request-URI` identifies a currently cached entity, that entity MUST be removed from the cache. Responses to this method are not cachable.

Caches that implement LINK should invalidate cached responses as defined in section 13.17 for PUT.

E.1.3 UNLINK

The UNLINK method removes one or more Link relationships from the existing resource identified by the `Request-URI`. These relationships may have been established using the LINK method or by any other method supporting the Link header. The removal of a link to a resource does not imply that the resource ceases to exist or becomes inaccessible for future references.

If the request passes through a cache and the `Request-URI` identifies a currently cached entity, that entity MUST be removed from the cache. Responses to this method are not cachable.

Caches that implement UNLINK should invalidate cached responses as defined in section 13.17 for PUT.

E.2 Additional Header Field Definitions

E.2.1 Content-Version

The `Content-Version` entity-header field defines the version tag associated with a rendition of an evolving entity. Together with the `Derived-From` field described in Section 10.18, it allows a group of people to work simultaneously on the creation of a work as an iterative process. The field SHOULD be used to allow evolution of a particular work along a single path. It SHOULD NOT be used to indicate derived works or renditions in different representations. It MAY also me used as an opaque value for comparing a cached entity's version with that of the current resource.

```
Content-Version = "Content-Version" ":" quoted-string
```

Examples of the Content-Version field include:

```
Content-Version: "2.1.2"

Content-Version: "Fred 19950116-12:26:48"

Content-Version: "2.5a4-omega7"
```

The value of the `Content-Version` field SHOULD be considered opaque to all parties but the origin server. A user agent MAY suggest a value for the version of an entity transferred via a PUT request; however, only the origin server can reliably assign that value.

E.2.2 Derived-From

The `Derived-From` entity-header field can be used to indicate the version tag of the resource from which the enclosed entity was derived before modifications were made by the sender. This field is used to help manage the process of merging successive changes to a resource, particularly when such changes are being made in parallel and from multiple sources.

```
Derived-From   = "Derived-From" ":" quoted-string
```

An example use of the field is:

```
Derived-From: "2.1.1"
```

The `Derived-From` field is required for `PUT` and `PATCH` requests if the entity being sent was previously retrieved from the same URI and a `Content-Version` header was included with the entity when it was last retrieved.

E.2.3 Link

The `Link` entity-header field provides a means for describing a relationship between two resources, generally between the requested resource and some other resource. An entity MAY include multiple Link values. Links at the metainformation level typically indicate relationships like hierarchical structure and navigation paths. The `Link` field is semantically equivalent to the <LINK> element in HTML [5].

```
Link           = "Link" ":" #("<" URI ">" *( ";" link-param )

link-param     = ( ( "rel" "=" relationship )
                   | ( "rev" "=" relationship )
                   | ( "title" "=" quoted-string )
                   | ( "anchor" "=" <"> URI <"> )

                   | ( link-extension ) )

link-extension = token [ "=" ( token | quoted-string ) ]


relationship   = sgml-name
               | ( <"> sgml-name *( SP sgml-name) <"> )


sgml-name      = ALPHA *( ALPHA | DIGIT | "." | "-" )
```

Relationship values are case-insensitive and MAY be extended within the constraints of the sgml-name syntax. The title parameter MAY be used to label the destination of a link such that it can be used as identification within a human-readable menu. The anchor parameter MAY be used to indicate a source anchor other than the entire current resource, such as a fragment of this resource or a third resource.

Examples of usage include:

```
Link: <http://www.cern.ch/TheBook/chapter2>; rel="Previous"

Link: <mailto:timbl@w3.org>; rev="Made"; title="Tim Berners-Lee"
```

The first example indicates that chapter2 is previous to this resource in a logical navigation path. The second indicates that the person responsible for making the resource available is identified by the given e-mail address.

E.2.4 URI

The URI header field has, in past versions of this specification, been used as a combination of the existing `Location`, `Content-Location`, and Alternates header fields. Its primary purpose has been to include a list of additional URIs for the resource, including names and mirror locations. However, it has become clear that the combination of many different functions within this single field has been a barrier to consistently and correctly implementing any of those functions. Furthermore, we believe that the identification of names and mirror locations would be better performed via the Link header field. The URI header field is therefore deprecated in favor of those other fields.

```
URI-header    = "URI" ":" 1#( "<" URI ">" )
```

E.2.5 Compatibility with HTTP/1.0 Persistent Connections

Some clients and servers may wish to be compatible with some previous implementations of persistent connections in HTTP version 1.0 clients and servers.

When connecting to an origin server an HTTP client MAY send the `Keep-Alive` connection-token in addition to the Persist connection-token:

```
Connection: Keep-Alive,Persist
```

An HTTP/1.0 server would then respond with the `Keep-Alive` connection token and the client may proceed with an HTTP/1.0 (or `Keep-Alive`) persistent connection.

An HTTP/1.1 server may also establish persistent connections with HTTP/1.0 clients upon receipt of a `Keep-Alive` connection token.

A persistent connection based on the `Keep-Alive` connection token MUST only use the "`Content-Length`" technique for marking the ending boundaries of entity-bodies. It MAY use pipe-lining.

A client MUST NOT send the `Keep-Alive` connection token to a proxy server as HTTP/1.0 proxy servers do not obey the rules of HTTP/1.1 for parsing the Connection header field.

E.2.5.1 The Keep-Alive Header

When the `Keep-Alive` connection-token has been transmitted with a request or a response a `Keep-Alive` header field MAY also be included. The `Keep-Alive` header field takes the following form:

```
Keep-Alive-header = "Keep-Alive" ":" 0# keepalive-param

keepalive-param = param-name "=" value
```

The `Keep-Alive` header itself is optional, and is used only if a parameter is being sent. HTTP/1.1 does not define any parameters.

If the `Keep-Alive` header is sent, the corresponding connection token MUST be transmitted. The `Keep-Alive` header MUST be ignored if received without the connection token.

F.1 Compatibility with Previous Versions

It is beyond the scope of a protocol specification to mandate compliance with previous versions. HTTP /1.1 was deliberately designed, however, to make supporting previous versions easy. While we are contemplating a separate document containing advice to implementers, we feel it worth noting that at the time of composing this specification, we would expect commercial HTTP/1.1 servers to::

- recognize the format of the Request-Line for HTTP/0.9, 1.0, and 1.1 requests;
- understand any valid request in the format of HTTP/0.9, 1.0, or 1.1;

- respond appropriately with a message in the same major version used by the client.

And we would expect HTTP/1.1 clients to:

- recognize the format of the Status-Line for HTTP/1.0 and 1.1 responses;
- understand any valid response in the format of HTTP/0.9, 1.0, or 1.1.

For most implementations of HTTP/1.0, each connection is established by the client prior to the request and closed by the server after sending the response. A few implementations implement the Keep-Alive version of persistent connections described in Section E.2.5.1.

G. Proxy Cache Implementation Guidelines

G.1 Support for Content Negotiation by Proxy Caches

The material in appendix G should go into a separate implementation guide as an informational RFC, rather than in this specification. (since it mostly describes 3 possible cache implementation strategies possible within the protocol, rather than just the two protocol facilities (transparent and opaque)). For the purposes of this (02) draft, we will leave it in as an appendix as it clarifies some points of how caching might work in the context of the HTTP/1.1 protocol.

If a resource is varying, this has an important effect on cache management, particularly for caching proxies which service a diverse set of user agents. Such proxy caches must correctly handle requests on varying resources in order not to disturb the negotiation process.

This specification distinguishes six levels of correct support for content negotiation by proxy caches. The text below describes these levels, but does not exhaustively list all mechanisms associated with support on these levels. In particular, mechanisms for handling partial requests on varying resources are not discussed.

1. A proxy cache providing level 1 support will never store in cache memory responses from varying resources (such responses always include at least one `Vary` header or `Alternates` header). When receiving a request on a varying resource, the proxy will thus always forward the request towards an upstream server. A level 1 proxy cache never makes selection decisions itself.
2. A proxy cache providing level 2 support is able to maintain in cache memory a mapping from the varying resource URI to a set of 200 (OK) response messages. When receiving a request on the varying resource, the proxy will forward this request to an upstream server after including an `If-Invalid` request header field listing the `CVal` header values of the associated cached 200 responses, as described in Section 10.52. If a 304 (Not Modified) response is received from the upstream server, the proxy updates, with the 304 response headers, the stored 200 response which has the same `CVal` header field as the 304 response. It then passes either the updated 200 response or the 304 response on to its client, the choice depending on the presence and contents of an `If-Invalid` header in the original request. If a 200 response is received from the upstream server, the proxy will update the set of responses it has for the varying resource by using the cache update algorithm described in Section 13.20, and pass on the 200 response to its client.
3. A proxy cache providing level 3 support is able to maintain in cache memory a mapping from the varying resource URI to a set of 200 (OK) response messages. In addition, it is able to maintain, for each cached 200 response belonging to the varying resource, a list of selecting request header sequences. This list of selecting request header sequences starts with the sequence taken from the request which initially caused an upstream server to return the cached 200 response, and continues with any additional sequences taken from subsequent requests which caused an upstream server to return a 304 response with a `CVal` header identical to the `CVal` header of that cached 200 response. When receiving a request on the varying resource, the proxy will iterate over all cached, fresh 200 responses associated with the resource. For each fresh 200

response, it will search the associated list of selecting request header sequences to see if a match to the headers of the current request can be found. If a match is found, the proxy will return the fresh 200 response in question. If no match is found, the proxy will switch to level 1 behavior and pass on the request to an upstream server. The response received from the upstream server may refresh a stale 200 response that was cached for the varying resource a side effect. **XXX previous sentence doesn't make sense…**

4. A proxy cache providing level 4 support provides transparent negotiation services for transparently negotiated resources, and provides level 1 support for opaquely negotiated resources.
5. A proxy cache providing level 5 support provides transparent negotiation services for transparently negotiated resources, and provides level 2 support for opaquely negotiated resources.
6. A proxy cache providing level 6 support provides transparent negotiation services for transparently negotiated resources, and provides level 3 support for opaquely negotiated resources.

Note: Implementation of support levels 4 to 6 is only possible when the planned content negotiation specification [29] is completed. The level numbers above were assigned to reflect expected caching efficiency in an environment where the proxy cache is serving a diverse set of clients. It is expected that level 4 proxies will be easier to implement than level 3 proxies.

G.2 Propagation of Changes in Opaque Selection

Level 3 and level 6 proxy caches not only cache the responses from an opaquely varying resource, they also cache the mappings from request headers to particular entities computed by the opaque selection algorithm located at the origin server. If this selection algorithm is changed by the resource author, for example because a Spanish text entity is added to a resource which previously only had English and French entities available, it is important to make the level 3 and 6 caches refresh their cached mappings. This can be done by changing the `CVal` header fields sent along with the original English and French responses. This change will eventually cause the proxies to replace the old English and French responses in cache memory, along with their associated lists of selecting request header sequences, by `new' English and French responses with fresh lists of selecting request header sequences. In order to guarantee an upper time bound for this update process, the resource author can include an appropriate `Cache-control: max-age=...` directive in the responses from the varying resource.

G.3 SLUSHY: State

This should probably be in the cookie ID, and not in this document at all.

HTTP implementations often support facilities for state management, often called "cookies"[35]. Cookies can not be cached by public (shared) caches, but since public documents may make up part of a "stateful dialog," and in particular the first document in a stateful dialog may be (for example) a public and cachable home page, servers that wish to receive the client's cookie on each request, or to issue a new cookie on requests for a document, must set the document up to require validation on each request (`Cache-Control: must-revalidate`)

In general, the cache control headers for responses control what a proxy has to do. If a document is fresh in a cache, a request containing a cookie does not have to be forwarded to the origin server, since (by definition) if the document can be served from a cache the origin server must have said there are no important side effects at the origin relating to requests for that document, and so, no changes to the cookie.

One important state issue bearing on caching is that for conditional requests that go through to the origin server, for which the origin server responds with 304 and also with a set-cookie header, caches must splice the set-cookie sent by the origin server into their own response. For example, this allows a home page to be cached, but stale, so that the only traffic to the origin server is to validate the home page, receiving a 304 and potentially a new cookie.

G.4 FLUID: Cache Replacement Algorithms

TBS

Should go into an implementers Informational RFC.

G.5 FLUID: Bypassing in Caching Hierarchies

This should also go into an implementers Informational RFC, and become grist for HTML's mill.

Many HTTP caching configurations involve hierarchies of caches, often designed to reduce bandwidth requirements rather than improving latency. However, if a cache at a low level in the hierarchy is sure that the cache(s) above it do not contain a cache entry to match a given request, that low-level cache can transmit the request directly to the origin server. This improves retrieval latency without increasing total bandwidth requirements (it even eliminates some packet transmissions) and is entirely appropriate for resources whose values are explicitly not cached.

We call this technique "request bypassing." Note that although the bypassing decision might be done by the ultimate client, in many cases the use of firewalls or unsophisticated clients means that the decision must be made by an intermediate-level cache.

In order for request bypassing to work in the most efficient possible way, the caches must be able to determine from the request whether the response is likely to be cachable. (It is important to err on the side of assuming cachability, since the assuming converse could seriously reduce the effectiveness of the higher-level caches.)

The current HTTP/1.1 draft specification does not include a foolproof mechanism to mark requests in this way. While we generally do not allow caching of responses to GET requests for URLs with a "?" in the rel_path part (see section 13.16), we also allow the origin server to mark responses to such queries as cachable. Therefore, any bypassing done using this heuristic runs the risk of giving up perfectly good opportunities to cache some resources.

XXX we have discussed various approaches for marking requests, all of which apparently require some kind of change to HTML to allow the origin server to pass the marks to the ultimate client. Some people suggest using special methods that are explicitly always cachable ("POST_WITH_NO_SIDE_EFFECTS", or more concisely "POSTC") or never cachable ("GET_QUERY", or more concisely "GETQ"). Others have suggested adding tags to HTML that would cause subsequent requests to carry some special sort of header. Neither solution has resulted in a consensus.

An origin server would be able to use POSTC only withHTTP/1.1 clients and proxies, and so would have to return different HTML forms depending on the protocol version in the request header. This would also imply using the proposed Vary: header with some token that indicates "varies based on request HTTP version," since we don't want a cache returning one of these HTML responses to an HTTP/1.0 client