Network Working Group
Request for Comments: 2616
Obsoletes: 2068
Category: Standards Track

R. Fielding
UC Irvine
J. Gettys
Compaq/W3C
J. C. Mogul
Compaq
H. Frystyk
W3C/MIT
L. Masinter
Xerox
P. Leach
Microsoft
T. Berners-Lee
W3C/MIT
June, 1999

Hypertext Transfer Protocol -- HTTP/1.1

Status of this Memo

This document specifies an Internet standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Internet Official Protocol Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

Copyright Notice

Abstract

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, protocol which can be used for many tasks beyond its use for hypertext, such as name servers and distributed object management systems, through extension of its request methods, error codes and headers [47]. 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", and is an update to RFC 2068 [33].

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 1945 [6], improved the protocol by allowing messages to be in the format of MIME-like messages, containing metainformation about the data transferred and modifiers on the request/response semantics. However, HTTP/1.0 does not sufficiently take into consideration the effects of hierarchical proxies, caching, the need for persistent connections, or virtual hosts. 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 includes more stringent requirements than HTTP/1.0 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 and headers that indicate the purpose of a request [47]. 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 to which a method is to be applied. Messages are passed in a format similar to that used by Internet mail [9] as defined by 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 systems, including those supported by the SMTP [16], NNTP [13], FTP [18], Gopher [2], and WAIS [10] protocols. In this way, HTTP allows basic hypermedia access to resources available from diverse applications.

1.2 Requirements

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [34].

An implementation is not compliant if it fails to satisfy one or more of the MUST or REQUIRED level requirements for the protocols it implements. An implementation that satisfies all the MUST or REQUIRED level and all the SHOULD level requirements for its protocols is said to be "unconditionally compliant"; one that satisfies all the MUST level requirements but not all the SHOULD level 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 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, as defined in section 3.2. Resources may be available in multiple representations (e.g. multiple languages, data formats, size, and resolutions) or vary in other ways.

entity
: The information transferred as the payload of a request or response. An entity consists of metainformation in the form of entity-header fields and content in the form of an entity-body, as described in section 7.

representation
: An entity included with a response that is subject to content negotiation, as described in section 12. There may exist multiple representations associated with a particular response status.

content negotiation
: The mechanism for selecting the appropriate representation when servicing a request, as described in section 12. The representation of entities in any response can be negotiated (including error responses).

variant
: A resource may have one, or more than one, representation(s) associated with it at any given instant. Each of these representations is termed a 'variant.' Use of the term 'variant' does not necessarily imply that the resource is subject to content negotiation.

client
: A 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. 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.

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 on, with possible translation, to other servers. A proxy MUST implement both the client and server requirements of this specification. A "transparent proxy" is a proxy that does not modify the request or response beyond what is required for proxy authentication and identification. A "non-transparent proxy" is a proxy that modifies the request or response in order to provide some added service to the user agent, such as group annotation services, media type transformation, protocol reduction, or anonymity filtering. Except where either transparent or non-transparent behavior is explicitly stated, the HTTP proxy requirements apply to both types of proxies.

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.

tunnel
: 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.

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

cacheable
: A response is cacheable if a cache is allowed to store a copy of the response message for use in answering subsequent requests. The rules for determining the cacheability of HTTP responses are defined in section 13. Even if a resource is cacheable, there may be additional constraints on whether a cache can use the cached copy for a particular request.

first-hand
: A response is first-hand if it comes directly and without unnecessary delay from the origin server, perhaps via one or more proxies. A response is also first-hand 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 sent 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 exceeded its freshness lifetime.

stale
: A response is stale if its age has passed its freshness lifetime.

semantically transparent

A cache behaves in a “semantically transparent” manner, with respect to a particular response, when its use affects neither the requesting client nor the origin server, except to improve performance. When a cache is semantically transparent, the client receives exactly the same response (except for hop-by-hop headers) that it would have received had its request been handled directly by the origin server.

validator

A protocol element (e.g., an entity tag or a Last-Modified time) that is used to find out whether a cache entry is an equivalent copy of an entity.

upstream/downstream

Upstream and downstream describe the flow of a message: all messages flow from upstream to downstream.

inbound/outbound

Inbound and outbound refer to the request and response paths for messages: “inbound” means “traveling toward the origin server”, and “outbound” means “traveling toward the user agent”

1.4 Overall Operation

The HTTP protocol is a request/response protocol. 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 entity-body content. The relationship between HTTP and MIME is described in appendix 19.4.

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 part 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 will 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 usefully cacheable, and some requests may contain modifiers which place special requirements on cache behavior. HTTP requirements for cache behavior and cacheable responses are defined in section 13.

In fact, there are a wide variety of architectures and configurations of caches and proxies currently being experimented with or deployed across the World Wide Web. These systems include national hierarchies of proxy caches to save transoceanic bandwidth, systems that broadcast or multicast cache entries, organizations that distribute subsets of cached data via CD-ROM, and so on. HTTP systems are used in corporate intranets over high-bandwidth links, and for access via PDAs with low-power radio links and intermittent connectivity. The goal of HTTP/1.1 is to support the wide diversity of configurations already deployed while introducing protocol constructs that meet the needs of those who build web applications that require high reliability and, failing that, at least reliable indications of failure.

HTTP communication usually 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.

In HTTP/1.0, most implementations used a new connection for each request/response exchange. In HTTP/1.1, a connection may be used for one or more request/response exchanges, although connections may be closed for a variety of reasons (see section 8.1).

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]. Implementors 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. White space 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 white space (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 white space (LWS) can be included between any two adjacent words (token or quoted-string), and between adjacent words and separators, without changing the interpretation of a field. At least one delimiter (LWS and/or separators) MUST exist between any two tokens (for the definition of "token" below), since they would otherwise be interpreted as a single token.

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 ANSI X3.4-1986 [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 sequence CR LF as the end-of-line marker for all protocol elements except the entity-body (see appendix 19.3 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 header field values can be folded onto multiple lines if the continuation line begins with a space or horizontal tab. All linear white space, including folding, has the same semantics as SP. A recipient MAY replace any linear white space with a single SP before interpreting the field value or forwarding the message downstream.

```
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 characters from character sets other than ISO-8859-1 [22] only when encoded according to the rules of RFC 2047 [14].

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

A CRLF is allowed in the definition of TEXT only as part of a header field continuation. It is expected that the folding LWS will be replaced with a single SP before interpretation of the TEXT value.

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 (as defined in section 3.6).

```
token          = 1*<any CHAR except CTLs or separators>
separators     = "(" | ")" | "<" | ">" | "@"
               | "," | ";" | ":" | "\" | <">
               | "/" | "[" | "]" | "?" | "="
               | "{" | "}" | 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 | quoted-pair | 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 | quoted-pair ) <"> )
qdtext         = <any TEXT except <">>
```

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. See RFC 2145 [36] for a fuller explanation.

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

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

Note that the major and minor numbers MUST 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 MUST be ignored by recipients and MUST NOT be sent.

An application that sends a request or response message that includes HTTP-Version of “HTTP/1.1” MUST be at least conditionally compliant with this specification. Applications that are at least conditionally compliant with this specification SHOULD use an HTTP-Version of “HTTP/1.1” in their messages, and MUST do so for any message that is not compatible with HTTP/1.0. For more details on when to send specific `HTTP-Version` values, see RFC 2145 [36].

The HTTP version of an application is the highest HTTP version for which the application is at least conditionally compliant.

Proxy and gateway applications need to be careful when forwarding messages in protocol versions different from that of the application. Since the protocol version indicates the protocol capability of the sender, a proxy/gateway MUST NOT send a message with a version indicator which is greater than its actual version. If a higher version request is received, the proxy/gateway MUST either downgrade the request version, or respond with an error, or switch to tunnel behavior.

Due to interoperability problems with HTTP/1.0 proxies discovered since the publication of RFC 2068[33], caching proxies MUST, gateways MAY, and tunnels MUST NOT upgrade the request to the highest version they support. The proxy/gateway's response to that request MUST be in the same major version as the request.

> Note: Converting between versions of HTTP may involve modification of header fields required or forbidden by the versions involved.

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 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. For definitive information on URL syntax and semantics, see "Uniform Resource Identifiers (URI): Generic Syntax and Semantics," RFC 2396 [42] (which replaces RFCs 1738 [4] and RFC 1808 [11]). This specification adopts the definitions of "`URI-reference`", "`absoluteURI`", "`relativeURI`", "`port`", "`host`","`abs_path`", "`rel_path`", and "`authority`" from that specification.

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 414 (Request-URI Too Long) status if a URI is longer than the server can handle (see section 10.4.15).

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

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 [ "?" query ]]
```

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` (section 5.1.2). The use of IP addresses in URLs 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). If a proxy receives a host name which is not a fully qualified domain name, it MAY add its domain to the host name it received. If a proxy receives a fully qualified domain name, the proxy MUST NOT change the host name.

3.2.3 URI Comparison

When comparing two URIs to decide if they match or not, a client SHOULD use a case-sensitive octet-by-octet comparison of the entire URIs, with these exceptions:

- A port that is empty or not given is equivalent to the default port for that `URI-reference`;
- 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 other than those in the “reserved” and “unsafe” sets (see RFC 2396 [42]) are equivalent to their “`"%" HEX HEX`” encoding.

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
```

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, obsoleted 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 (for compatibility with HTTP/1.0), though they MUST only generate the RFC 1123 format for representing HTTP-date values in header fields. See section 19.3 for further information.

> Note: Recipients of date values are encouraged to be robust in accepting date values that may have been sent 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 Greenwich Mean Time (GMT), without exception. For the purposes of HTTP, GMT is exactly equal to UTC (Coordinated Universal Time). This is indicated in the first two formats by the inclusion of “GMT” as the three-letter abbreviation for time zone, and MUST be assumed when reading the asctime format. HTTP-date is case sensitive and MUST NOT include additional LWS beyond that specifically included as SP in the grammar.

```
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.

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.

```
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 encoding, 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].

```
charset = 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.

Implementors should be aware of IETF character set requirements [38] [41].

3.4.1 Missing Charset

Some HTTP/1.0 software has interpreted a `Content-Type` header without charset parameter incorrectly to mean “recipient should guess.” Senders wishing to defeat this behavior MAY include a charset parameter even when the charset is ISO-8859-1 and SHOULD do so when it is known that it will not confuse the recipient.

Unfortunately, some older HTTP/1.0 clients did not deal properly with an explicit charset parameter. HTTP/1.1 recipients MUST respect the charset label provided by the sender; and those user agents that have a provision to “guess” a charset MUST use the charset from the content-type field if they support that charset, rather than the recipient’s preference, when initially displaying a document. See section 3.7.1.

3.5 Content Codings

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

```
content-coding   = token
```

All content-coding values are case-insensitive. HTTP/1.1 uses content-coding values in the `Accept-Encoding` (section 14.3) and `Content-Encoding` (section 14.11) 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.
The Internet Assigned Numbers Authority (IANA) acts as a registry for content-coding value tokens. Initially, the registry contains the following tokens:

`gzip` An encoding format produced by the file compression program "gzip" (GNU zip) as described in RFC 1952 [25]. This format is a Lempel-Ziv coding (LZ77) with a 32 bit CRC.

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

Use of program names for the identification of encoding formats is not desirable and is discouraged for future encodings. Their use here is representative of historical practice, not good design. For compatibility with previous implementations of HTTP, applications SHOULD consider "x-gzip" and "x-compress" to be equivalent to "gzip" and "compress" respectively.

`deflate`
The "zlib" format defined in RFC 1950 [31] in combination with the "deflate" compression mechanism described in RFC 1951 [29].

`identity`
The default (identity) encoding; the use of no transformation whatsoever. This content-coding is used only in the `Accept-Encoding` header, and SHOULD NOT be used in the `Content-Encoding` header.

New content-coding value tokens SHOULD be registered; to allow interoperability between clients and servers, specifications of the content coding algorithms needed to implement a new value SHOULD be publicly available and adequate for independent implementation, and 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 entity.

```
transfer-coding         = "chunked" | transfer-extension
transfer-extension      = token *( ";" parameter )
```

Parameters are in the form of attribute/value pairs.

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

All transfer-coding values are case-insensitive. HTTP/1.1 uses transfer-coding values in the `TE` header field (section 14.39) and in the `Transfer-Encoding` header field (section 14.41).

Whenever a transfer-coding is applied to a message-body, the set of transfer-codings MUST include "`chunked`", unless the message is terminated by closing the connection. When the "`chunked`" transfer-coding is used, it MUST be the last transfer-coding applied to the message-body. The "`chunked`" transfer-coding MUST NOT be applied more than once to a message-body. These rules allow the recipient to determine the transfer-length of the message (section 4.4).

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.

The Internet Assigned Numbers Authority (IANA) acts as a registry for transfer-coding value tokens. Initially, the registry contains the following tokens: "`chunked`" (section 3.6.1), "`identity`" (section 3.6.2), "`gzip`" (section 3.5), "`compress`" (section 3.5), and "`deflate`" (section 3.5).

New transfer-coding value tokens SHOULD be registered in the same way as new content-coding value tokens (section 3.5).

A server which receives an entity-body with a transfer-coding it does not understand SHOULD return 501 (Unimplemented), and close the connection. A server MUST NOT send transfer-codings to an HTTP/1.0 client.

3.6.1 Chunked Transfer Coding

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 trailer 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
                 last-chunk
                 trailer
                 CRLF
chunk          = chunk-size [ chunk-extension ] CRLF
                 chunk-data CRLF
chunk-size     = 1*HEX
last-chunk     = 1*("0") [ chunk-extension ] CRLF

chunk-extension= *( ";" chunk-ext-name [ "=" chunk-ext-val ] )
chunk-ext-name = token
chunk-ext-val  = token | quoted-string
chunk-data     = chunk-size(OCTET)
trailer        = *(entity-header CRLF)
```

The `chunk-size` field is a string of hex digits indicating the size of the chunk. The chunked encoding is ended by any chunk whose size is zero, followed by the trailer, which is terminated by an empty line.

The trailer allows the sender to include additional HTTP header fields at the end of the message. The `Trailer` header field can be used to indicate which header fields are included in a trailer (see section 14.40).

A server using chunked transfer-coding in a response MUST NOT use the trailer for any header fields unless at least one of the following is true:

a) the request included a `TE` header field that indicates “trailers” is acceptable in the transfer-coding of the response, as described in section 14.39; or,

b) the server is the origin server for the response, the trailer fields consist entirely of optional metadata, and the recipient could use the message (in a manner acceptable to the origin server) without receiving this metadata. In other words, the origin server is willing to accept the possibility that the trailer fields might be silently discarded along the path to the client.

This requirement prevents an interoperability failure when the message is being received by an HTTP/1.1 (or later) proxy and forwarded to an HTTP/1.0 recipient. It avoids a situation where compliance with the protocol would have necessitated a possibly infinite buffer on the proxy.

An example process for decoding a `Chunked-Body` is presented in appendix 19.4.6.

All HTTP/1.1 applications MUST be able to receive and decode the “chunked” transfer-coding, and MUST ignore `chunk-extension` extensions they do not understand.

3.7 Media Types

HTTP uses Internet Media Types [17] in the `Content-Type` (section 14.17) and `Accept` (section 14.1) 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 (as defined in section 3.6).

The type, subtype, and parameter attribute names are case-insensitive. Parameter values might or might not be case-sensitive, depending on the semantics of the parameter name. Linear white space (`LWS`) MUST NOT be used between the type and subtype, nor between an attribute and its value. The presence or absence of a parameter might be significant to the processing of a media-type, depending on its definition within the media type registry.

Note that some older HTTP applications do not recognize media type parameters. When sending data to older HTTP applications, implementations SHOULD only use media type parameters when they are required by that type/subtype definition.

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. An entity-body transferred via HTTP messages MUST be represented in the appropriate canonical form prior to its transmission except for "text" types, as defined in the next paragraph.

When in canonical form, media subtypes of the "text" type use `CRLF` as the text line break. HTTP relaxes this requirement and allows the transport of text media with plain `CR` or `LF` alone representing a line break when it is done consistently for an entire 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 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` MUST NOT be substituted for `CRLF` within any of the HTTP control structures (such as header fields and multipart boundaries).

If an entity-body is encoded with a content-coding, the underlying data MUST be in a form defined above prior to being encoded.

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. See section 3.4.1 for compatibility problems.

3.7.2 Multipart Types

MIME provides for a number of "multipart" types -- encapsulations of one or more entities within a single message-body. All multipart types share a common syntax, as defined in section 5.1.1 of RFC 2046 [40], 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 2046, the epilogue of any multipart message MUST be empty; HTTP applications MUST NOT transmit the epilogue (even if the original multipart contains an epilogue). These restrictions exist in order to preserve the self-delimiting nature of a multipart message-body, wherein the "end" of the message-body is indicated by the ending multipart boundary.

In general, HTTP treats a multipart message-body no differently than any other media type: strictly as payload. The one exception is the "multipart/byteranges" type (appendix 19.2) when it appears in a 206 (Partial Content) response, which will be interpreted by some HTTP caching mechanisms as described in sections 13.5.4 and 14.16. In all other cases, an HTTP user agent SHOULD follow the same or similar behavior as a MIME user agent would upon receipt of a multipart type. The MIME header fields within each body-part of a multipart message-body do not have any significance to HTTP beyond that defined by their MIME semantics.

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 by software name and version. Most fields using product tokens also allow sub-products which form a significant part of the application to be listed, separated by white space. 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. They MUST NOT be used for advertising or other non-essential information. 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. A weight is normalized to a real number in the range 0 through 1, where 0 is the minimum and 1 the maximum value. If a parameter has a quality value of 0, then content with this parameter is ‘not acceptable’ for the client. 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 ] )
               | ( "1" [ "." 0*3("0") ] )
```

“Quality values” is a misnomer, since these values merely represent relative degradation in desired 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
```

White space is not allowed within the tag and all tags are case-insensitive. The name space 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; all but the last are examples of tags which could be registered in future.)

3.11 Entity Tags

Entity tags are used for comparing two or more entities from the same requested resource. HTTP/1.1 uses entity tags in the `ETag` (section 14.19), `If-Match` (section 14.24), `If-None-Match` (section 14.26), and `If-Range` (section 14.27) header fields. The definition of how they are used and compared as cache validators is in section 13.3.3. An entity tag consists of an opaque quoted string, possibly prefixed by a weakness indicator.

```
entity-tag = [ weak ] opaque-tag
weak       = "W/"
opaque-tag = quoted-string
```

A “strong entity tag” MAY be shared by two entities of a resource only if they are equivalent by octet equality.

A “weak entity tag,” indicated by the "W/" prefix, MAY be shared by two entities of a resource only if the entities are equivalent and could be substituted for each other with no significant change in semantics. A weak entity tag can only be used for weak comparison.

An entity tag MUST be unique across all versions of all entities associated with a particular resource. A given entity tag value MAY be used for entities obtained by requests on different URIs. The use of the same entity tag value in conjunction with entities obtained by requests on different URIs does not imply the equivalence of those entities.

3.12 Range Units

HTTP/1.1 allows a client to request that only part (a range of) the response entity be included within the response. HTTP/1.1 uses range units in the `Range` (section 14.35) and `Content-Range` (section 14.16) header fields. An entity can be broken down into subranges according to various structural units.

```
range-unit       = bytes-unit | other-range-unit
bytes-unit       = "bytes"
other-range-unit = token
```

The only range unit defined by HTTP/1.1 is “bytes”. HTTP/1.1 implementations MAY ignore ranges specified using other units. HTTP/1.1 has been designed to allow implementations of applications that do not depend on knowledge of ranges.

4 HTTP Message

4.1 Message Types

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

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

`Request` (section 5) and `Response` (section 6) messages use the generic message format of RFC 822 [9] for transferring entities (the payload of the message). Both types of message consist of a start-line, zero or more header fields (also known as “headers”), an empty line (i.e., a line with nothing preceding the CRLF) indicating the end of the header fields, and possibly a message-body.

```
generic-message = start-line
                  *(message-header CRLF)
                  CRLF
                  [ message-body ]
start-line      = Request-Line | Status-Line
```

In the interest of robustness, servers SHOULD ignore any empty line(s) received where a `Request-Line` is expected. In other words, if the server is reading the protocol stream at the beginning of a message and receives a CRLF first, it should ignore the CRLF.

Certain buggy HTTP/1.0 client implementations generate extra CRLF’s after a POST request. To restate what is explicitly forbidden by the BNF, an HTTP/1.1 client MUST NOT preface or follow a request with an extra CRLF.

4.2 Message Headers

HTTP header fields, which include general-header (section 4.5), request-header (section 5.3), 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. Applications ought to follow "common form", where one is known or indicated, when generating HTTP constructs, since there might exist some implementations that fail to accept anything beyond the common forms.

```
message-header = field-name ":" [ field-value ]
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, separators, and quoted-string>
```

The field-content does not include any leading or trailing LWS: linear white space occurring before the first non-whitespace character of the field-value or after the last non-whitespace character of the field-value. Such leading or trailing LWS MAY be removed without changing the semantics of the field value. Any LWS that occurs between field-content MAY be replaced with a single SP before interpreting the field value or forwarding the message downstream.

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 message-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. The order in which header fields with the same field-name are received is therefore significant to the interpretation of the combined field value, and thus a proxy MUST NOT change the order of these field values when a message is forwarded.

4.3 Message Body

The message-body (if any) of an HTTP message is used to carry the entity-body associated with the request or response. The message-body differs from the entity-body only when a transfer-coding has been applied, as indicated by the Transfer-Encoding header field (section 14.41).

```
message-body = entity-body
             | <entity-body encoded as per Transfer-Encoding>
```

Transfer-Encoding MUST be used to indicate any transfer-codings applied by an application to ensure safe and proper transfer of the message. Transfer-Encoding is a property of the message, not of the entity, and thus MAY be added or removed by any application along the request/response chain. (However, section 3.6 places restrictions on when certain transfer-codings may be used.)

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

The presence of a message-body in a request is signaled by the inclusion of a Content-Length or Transfer-Encoding header field in the request's message-headers. A message-body MUST NOT be included in a request if the specification of the request method (section 5.1.1) does not allow sending an entity-body in requests. A server SHOULD read and forward a message-body on any request; if the request method does not include defined semantics for an entity-body, then the message-body SHOULD be ignored when handling the request.

For response messages, whether or not a message-body is included with a message is dependent on both the request method and the response status code (section 6.1.1). All responses to the HEAD request method MUST NOT include a message-body, even though the presence of entity-header fields might lead one to believe they do. All 1xx (informational), 204 (no content), and 304 (not modified) responses MUST NOT include a message-body. All other responses do include a message-body, although it MAY be of zero length.

4.4 Message Length

The transfer-length of a message is the length of the message-body as it appears in the message; that is, after any transfer-codings have been applied. When a message-body is included with a message, the transfer-length of that body is determined by one of the following (in order of precedence):

1. Any response message which "MUST NOT" include a message-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.

2. If a `Transfer-Encoding` header field (section 14.41) is present and has any value other than "`identity`", then the transfer-length is defined by use of the "`chunked`" transfer-coding (section 3.6), unless the message is terminated by closing the connection.

3. If a `Content-Length` header field (section 14.13) is present, its decimal value in OCTETs represents both the entity-length and the transfer-length. The `Content-Length` header field MUST NOT be sent if these two lengths are different (i.e., if a `Transfer-Encoding` header field is present). If a message is received with both a `Transfer-Encoding` header field and a `Content-Length` header field, the latter MUST be ignored.

4. If the message uses the media type "multipart/byteranges", and the transfer-length is not otherwise specified, then this self-delimiting media type defines the transfer-length. This media type MUST NOT be used unless the sender knows that the recipient can parse it; the presence in a request of a `Range` header with multiple byte-range specifiers from a 1.1 client implies that the client can parse multipart/byteranges responses.

   A range header might be forwarded by a 1.0 proxy that does not understand multipart/byteranges; in this case the server MUST delimit the message using methods defined in items 1,3 or 5 of this section.

5. By the server closing the connection. (Closing the connection cannot be used to indicate the end of a request body, since that would leave no possibility for the server to send back a response.)

For compatibility with HTTP/1.0 applications, HTTP/1.1 requests containing a message-body MUST include a valid `Content-Length` header field unless the server is known to be HTTP/1.1 compliant. If a request contains a message-body and a `Content-Length` is not given, the server SHOULD respond with 400 (bad request) if it cannot determine the length of the message, or with 411 (length required) if it wishes to insist on receiving a valid `Content-Length`.

All HTTP/1.1 applications that receive entities MUST accept the "chunked" transfer-coding (section 3.6), thus allowing this mechanism to be used for messages when the message length cannot be determined in advance.

Messages MUST NOT include both a `Content-Length` header field and a non-identity transfer-coding. If the message does include a non-identity transfer-coding, the `Content-Length` MUST be ignored.

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

4.5 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 header fields apply only to the message being transmitted.

```
general-header = Cache-Control            ; Section 14.9
               | Connection               ; Section 14.10
               | Date                     ; Section 14.18
               | Pragma                   ; Section 14.32
               | Trailer                  ; Section 14.40
               | Transfer-Encoding        ; Section 14.41
```

```
                      | Upgrade                  ; Section 14.42
                      | Via                      ; Section 14.45
                      | Warning                  ; Section 14.46
```

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.

```
 Request       = Request-Line              ; Section 5.1
                 *(( general-header        ; Section 4.5
                    | request-header       ; Section 5.3
                    | entity-header ) CRLF)  ; Section 7.1
                 CRLF
                 [ message-body ]          ; Section 4.3
```

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 is 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 9.2
               | "GET"                    ; Section 9.3
               | "HEAD"                   ; Section 9.4
               | "POST"                   ; Section 9.5
               | "PUT"                    ; Section 9.6
               | "DELETE"                 ; Section 9.7
               | "TRACE"                  ; Section 9.8
               | "CONNECT"                ; Section 9.9
               | extension-method
extension-method = token
```

The list of methods allowed by a resource can be specified in an Allow header field (section 14.7). The return code of the response always notifies the client whether a method is currently allowed on a resource, since the set of allowed methods can change dynamically. An origin server SHOULD return the status code 405 (Method Not Allowed) if the method is known by the origin server but not allowed for the requested resource, and 501 (Not Implemented) if the method is unrecognized or not implemented by the origin server. The methods GET and HEAD MUST be supported by all general-purpose servers. 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 9.

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 | authority
```

The four 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 REQUIRED when the request is being made to a proxy. The proxy is requested to forward the request or service it from a valid cache, and return the response. 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
```

To allow for transition to absoluteURIs in all requests in future versions of HTTP, all HTTP/1.1 servers MUST accept the absoluteURI form in requests, even though HTTP/1.1 clients will only generate them in requests to proxies.

The authority form is only used by the CONNECT method (section 9.9).

The most common form of Request-URI is that used to identify a resource on an origin server or gateway. In this case the absolute path of the URI MUST be transmitted (see section 3.2.1, abs_path) as the Request-URI, and the network location of the URI (authority) MUST be transmitted in a Host header field. 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 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).

The Request-URI is transmitted in the format specified in section 3.2.1. If the Request-URI is encoded using the "% HEX HEX" encoding [42], the origin server MUST decode the Request-URI in order to properly interpret the request. Servers SHOULD respond to invalid Request-URIs with an appropriate status code.

A transparent proxy MUST NOT rewrite the "abs_path" part of the received Request-URI when forwarding it to the next inbound server, except as noted above to replace a null abs_path with "/".

> Note: The "no rewrite" rule prevents the proxy from changing the meaning of the request when the origin server is improperly using a non-reserved URI character for a reserved purpose. Implementors should be aware that some pre-HTTP/1.1 proxies have been known to rewrite the Request-URI.

5.2 The Resource Identified by a Request

The exact resource identified by an Internet request is determined by examining both the Request-URI and the Host header field.

An origin server that does not allow resources to differ by the requested host MAY ignore the Host header field value when determining the resource identified by an HTTP/1.1 request. (But see section 19.6.1.1 for other requirements on Host support in HTTP/1.1.)

An origin server that does differentiate resources based on the host requested (sometimes referred to as virtual hosts or vanity host names) MUST use the following rules for determining the requested resource on an HTTP/1.1 request:

1. If Request-URI is an absoluteURI, the host is part of the Request-URI. Any Host header field value in the request MUST be ignored.

2. If the Request-URI is not an absoluteURI, and the request includes a Host header field, the host is determined by the Host header field value.

3. If the host as determined by rule 1 or 2 is not a valid host on the server, the response MUST be a 400 (Bad Request) error message.

Recipients of an HTTP/1.0 request that lacks a Host header field MAY attempt to use heuristics (e.g., examination of the URI path for something unique to a particular host) in order to determine what exact resource is being requested.

5.3 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 invocation.

```
request-header = Accept                   ; Section 14.1
               | Accept-Charset           ; Section 14.2
               | Accept-Encoding          ; Section 14.3
               | Accept-Language          ; Section 14.4
               | Authorization            ; Section 14.8
               | Expect                   ; Section 14.20
               | From                     ; Section 14.22
               | Host                     ; Section 14.23
               | If-Match                 ; Section 14.24
               | If-Modified-Since        ; Section 14.25
               | If-None-Match            ; Section 14.26
               | If-Range                 ; Section 14.27
               | If-Unmodified-Since      ; Section 14.28
               | Max-Forwards             ; Section 14.31
               | Proxy-Authorization      ; Section 14.34
               | Range                    ; Section 14.35
               | Referer                  ; Section 14.36
               | TE                       ; Section 14.39
               | User-Agent               ; Section 14.43
```

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 with an HTTP response message.

```
Response      = Status-Line               ; Section 6.1
                *(( general-header        ; Section 4.5
                 | response-header        ; Section 6.2
                 | entity-header ) CRLF)  ; Section 7.1
                CRLF
                [ message-body ]          ; Section 7.2
```

6.1 Status-Line

The first line of a 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. These codes are fully defined in section 10. 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 recommendations -- they MAY be replaced by local equivalents without affecting the protocol.

```
Status-Code    =
         "100"  ; Section 10.1.1: Continue
       | "101"  ; Section 10.1.2: Switching Protocols
       | "200"  ; Section 10.2.1: OK
       | "201"  ; Section 10.2.2: Created
       | "202"  ; Section 10.2.3: Accepted
       | "203"  ; Section 10.2.4: Non-Authoritative Information
       | "204"  ; Section 10.2.5: No Content
       | "205"  ; Section 10.2.6: Reset Content
       | "206"  ; Section 10.2.7: Partial Content
       | "300"  ; Section 10.3.1: Multiple Choices
       | "301"  ; Section 10.3.2: Moved Permanently
       | "302"  ; Section 10.3.3: Found
       | "303"  ; Section 10.3.4: See Other
       | "304"  ; Section 10.3.5: Not Modified
       | "305"  ; Section 10.3.6: Use Proxy
       | "307"  ; Section 10.3.8: Temporary Redirect
       | "400"  ; Section 10.4.1: Bad Request
       | "401"  ; Section 10.4.2: Unauthorized
       | "402"  ; Section 10.4.3: Payment Required
       | "403"  ; Section 10.4.4: Forbidden
       | "404"  ; Section 10.4.5: Not Found
       | "405"  ; Section 10.4.6: Method Not Allowed
       | "406"  ; Section 10.4.7: Not Acceptable
       | "407"  ; Section 10.4.8: Proxy Authentication Required
       | "408"  ; Section 10.4.9: Request Time-out
       | "409"  ; Section 10.4.10: Conflict
       | "410"  ; Section 10.4.11: Gone
       | "411"  ; Section 10.4.12: Length Required
       | "412"  ; Section 10.4.13: Precondition Failed
       | "413"  ; Section 10.4.14: Request Entity Too Large
       | "414"  ; Section 10.4.15: Request-URI Too Large
       | "415"  ; Section 10.4.16: Unsupported Media Type
       | "416"  ; Section 10.4.17: Requested range not satisfiable
       | "417"  ; Section 10.4.18: Expectation Failed
       | "500"  ; Section 10.5.1: Internal Server Error
       | "501"  ; Section 10.5.2: Not Implemented
       | "502"  ; Section 10.5.3: Bad Gateway
       | "503"  ; Section 10.5.4: Service Unavailable
       | "504"  ; Section 10.5.5: Gateway Time-out
       | "505"  ; Section 10.5.6: 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 = Accept-Ranges           ; Section 14.5
                | Age                     ; Section 14.6
                | ETag                    ; Section 14.19
                | Location                ; Section 14.30
                | Proxy-Authenticate      ; Section 14.33
                | Retry-After             ; Section 14.37
                | Server                  ; Section 14.38
                | Vary                    ; Section 14.44
                | WWW-Authenticate        ; Section 14.47
```

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

`Request` and `Response` messages MAY transfer an entity if not otherwise restricted by the request method or response status code. An entity consists of entity-header fields and an entity-body, although some responses will only include the entity-headers.

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 metainformation about the entity-body or, if no body is present, about the resource identified by the request. Some of this metainformation is OPTIONAL; some might be REQUIRED by portions of this specification.

```
entity-header  = Allow                    ; Section 14.7
               | Content-Encoding         ; Section 14.11
               | Content-Language         ; Section 14.12
               | Content-Length           ; Section 14.13
               | Content-Location         ; Section 14.14
               | Content-MD5              ; Section 14.15
               | Content-Range            ; Section 14.16
               | Content-Type             ; Section 14.17
               | Expires                  ; Section 14.21
               | Last-Modified            ; Section 14.29
               | extension-header
extension-header = message-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 MUST be forwarded by transparent 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 only present in a message when a message-body is present, as described in section 4.3. The entity-body is obtained from the message-body by decoding any `Transfer-Encoding` that might have been applied to ensure safe and proper transfer of the message.

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` and `Content-Encoding`. These define a two-layer, ordered encoding model:

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

`Content-Type` specifies the media type of the underlying data. `Content-Encoding` may be used to indicate any additional content codings applied to the data, usually for the purpose of data compression, that are a property of the requested resource. There is no default encoding.

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` field, the recipient MAY attempt to guess the media type via inspection of its content and/or the name extension(s) of the URI used to identify the resource. If the media type remains unknown, the recipient SHOULD treat it as type "`application/octet-stream`".

7.2.2 Entity Length

The entity-length of a message is the length of the message-body before any transfer-codings have been applied. Section 4.4 defines how the transfer-length of a message-body is determined.

8 Connections

8.1 Persistent Connections

8.1.1 Purpose

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 require a client to make multiple requests of the same server in a short amount of time. Analysis of these performance problems and results from a prototype implementation are available [26] [30]. Implementation experience and measurements of actual HTTP/1.1 (RFC 2068) implementations show good results [39]. Alternatives have also been explored, for example, T/TCP [27].

Persistent HTTP connections have a number of advantages:

- By opening and closing fewer TCP connections, CPU time is saved in routers and hosts (clients, servers, proxies, gateways, tunnels, or caches), and memory used for TCP protocol control blocks can be saved in hosts.
- HTTP requests and responses can be pipelined on a connection. Pipelining 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.
- Latency on subsequent requests is reduced since there is no time spent in TCP's connection opening handshake.
- 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 is reported.

HTTP implementations SHOULD implement persistent connections.

8.1.2 Overall Operation

A significant difference between HTTP/1.1 and earlier versions of HTTP is that persistent connections are the default behavior of any HTTP connection. That is, unless otherwise indicated, the client SHOULD assume that the server will maintain a persistent connection, even after error responses from the server.

Persistent connections provide a mechanism by which a client and a server can signal the close of a TCP connection. This signaling takes place using the `Connection` header field (section 14.10). Once a close has been signaled, the client MUST NOT send any more requests on that connection.

8.1.2.1 Negotiation

An HTTP/1.1 server MAY assume that a HTTP/1.1 client intends to maintain a persistent connection unless a `Connection` header including the connection-token "`close`" was sent in the request. If the server chooses to close the connection immediately after sending the response, it SHOULD send a `Connection` header including the connection-token `close`.

An HTTP/1.1 client MAY expect a connection to remain open, but would decide to keep it open based on whether the response from a server contains a `Connection` header with the connection-token `close`. In case the client does not want to maintain a connection for more than that request, it SHOULD send a `Connection` header including the connection-token `close`.

If either the client or the server sends the `close` token in the `Connection` header, that request becomes the last one for the connection.

Clients and servers SHOULD NOT assume that a persistent connection is maintained for HTTP versions less than 1.1 unless it is explicitly signaled. See section 19.6.2 for more information on backward compatibility with HTTP/1.0 clients.

In order to remain persistent, all messages on the connection MUST have a self-defined message length (i.e., one not defined by closure of the connection), as described in section 4.4.

8.1.2.2 Pipelining

A client that supports persistent connections MAY "pipeline" its requests (i.e., send multiple requests without waiting for each response). A server MUST send its responses to those requests in the same order that the requests were received.

Clients which assume persistent connections and pipeline immediately after connection establishment SHOULD be prepared to retry their connection if the first pipelined attempt fails. If a client does such a retry, it MUST NOT pipeline before it knows the connection is persistent. Clients MUST also be prepared to resend their requests if the server closes the connection before sending all of the corresponding responses.

Clients SHOULD NOT pipeline requests using non-idempotent methods or non-idempotent sequences of methods (see section 9.1.2). Otherwise, a premature termination of the transport connection could lead to indeterminate results. A client wishing to send a non-idempotent request SHOULD wait to send that request until it has received the response status for the previous request.

8.1.3 Proxy Servers

It is especially important that proxies correctly implement the properties of the `Connection` header field as specified in section 14.10.

The proxy server MUST signal 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 HTTP/1.1 persistent connection with an HTTP/1.0 client (but see RFC 2068 [33] for information and discussion of the problems with the `Keep-Alive` header implemented by many HTTP/1.0 clients).

8.1.4 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 (or existence) 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 side's 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 might have started to send a new request at the same time that the server has decided to close the "idle" connection. From the server's point of view, the connection is being closed while it was idle, but from the client's point of view, a request is in progress.

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 sequence of requests without user interaction so long as the request sequence is idempotent (see section 9.1.2). Non-idempotent methods or sequences MUST NOT be automatically retried, although user agents MAY offer a human operator the choice of retrying the request(s). Confirmation by user-agent software with semantic understanding of the application MAY substitute for user confirmation. The automatic retry SHOULD NOT be repeated if the second sequence of requests 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.

Clients that use persistent connections SHOULD limit the number of simultaneous connections that they maintain to a given server. A single-user client SHOULD NOT maintain more than 2 connections with any server or 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.

8.2 Message Transmission Requirements

8.2.1 Persistent Connections and Flow Control

HTTP/1.1 servers SHOULD 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.

8.2.2 Monitoring Connections for Error Status Messages

An HTTP/1.1 (or later) client sending a message-body SHOULD monitor the network connection for an error status while it is transmitting the request. If the client sees an error status, it SHOULD immediately cease transmitting the body. If the body is being sent using a "chunked" encoding (section 3.6), a zero length chunk and empty trailer MAY be used to prematurely mark the end of the message. If the body was preceded by a `Content-Length` header, the client MUST close the connection.

8.2.3 Use of the 100 (Continue) Status

The purpose of the 100 (Continue) status (see section 10.1.1) is to allow a client that is sending a request message with a request body to determine if the origin server is willing to accept the request (based on the request headers) before the client sends the request body. In some cases, it might either be inappropriate or highly inefficient for the client to send the body if the server will reject the message without looking at the body.

Requirements for HTTP/1.1 clients:

- If a client will wait for a 100 (Continue) response before sending the request body, it MUST send an `Expect` request-header field (section 14.20) with the “100-continue” expectation.
- A client MUST NOT send an `Expect` request-header field (section 14.20) with the “100-continue” expectation if it does not intend to send a request body.

Because of the presence of older implementations, the protocol allows ambiguous situations in which a client may send “Expect: 100-continue” without receiving either a 417 (Expectation Failed) status or a 100 (Continue) status. Therefore, when a client sends this header field to an origin server (possibly via a proxy) from which it has never seen a 100 (Continue) status, the client SHOULD NOT wait for an indefinite period before sending the request body.

Requirements for HTTP/1.1 origin servers:

- Upon receiving a request which includes an `Expect` request-header field with the “100-continue” expectation, an origin server MUST either respond with 100 (Continue) status and continue to read from the input stream, or respond with a final status code. The origin server MUST NOT wait for the request body before sending the 100 (Continue) response. If it responds with a final status code, it MAY close the transport connection or it MAY continue to read and discard the rest of the request. It MUST NOT perform the requested method if it returns a final status code.
- An origin server SHOULD NOT send a 100 (Continue) response if the request message does not include an `Expect` request-header field with the “100-continue” expectation, and MUST NOT send a 100 (Continue) response if such a request comes from an HTTP/1.0 (or earlier) client. There is an exception to this rule: for compatibility with RFC 2068, a server MAY send a 100 (Continue) status in response to an HTTP/1.1 PUT or POST request that does not include an `Expect` request-header field with the “100-continue” expectation. This exception, the purpose of which is to minimize any client processing delays associated with an undeclared wait for 100 (Continue) status, applies only to HTTP/1.1 requests, and not to requests with any other HTTP-version value.
- An origin server MAY omit a 100 (Continue) response if it has already received some or all of the request body for the corresponding request.
- An origin server that sends a 100 (Continue) response MUST ultimately send a final status code, once the request body is received and processed, unless it terminates the transport connection prematurely.
- If an origin server receives a request that does not include an `Expect` request-header field with the “100-continue” expectation, the request includes a request body, and the server responds with a final status code before reading the entire request body from the transport connection, then the server SHOULD NOT close the transport connection until it has read the entire request, or until the client closes the connection. Otherwise, the client might not reliably receive the response message. However, this requirement is not be construed as preventing a server from defending itself against denial-of-service attacks, or from badly broken client implementations.

Requirements for HTTP/1.1 proxies:

- If a proxy receives a request that includes an `Expect` request-header field with the “100-continue” expectation, and the proxy either knows that the next-hop server complies with HTTP/1.1 or higher, or does not know the HTTP version of the next-hop server, it MUST forward the request, including the `Expect` header field.

- If the proxy knows that the version of the next-hop server is HTTP/1.0 or lower, it MUST NOT forward the request, and it MUST respond with a 417 (Expectation Failed) status.
- Proxies SHOULD maintain a cache recording the HTTP version numbers received from recently-referenced next-hop servers.
- A proxy MUST NOT forward a 100 (Continue) response if the request message was received from an HTTP/1.0 (or earlier) client and did not include an `Expect` request-header field with the "100-continue" expectation. This requirement overrides the general rule for forwarding of 1xx responses (see section 10.1).

8.2.4 Client Behavior if Server Prematurely Closes Connection

If an HTTP/1.1 client sends a request which includes a request body, but which does not include an `Expect` request-header field with the "100-continue" expectation, and if the client is not directly connected to an HTTP/1.1 origin server, and if the client sees the connection close before receiving any status from the server, the client SHOULD retry the request. If the client does retry this request, it MAY use the following "binary exponential backoff" algorithm to be assured of obtaining a reliable response:

1. Initiate a new connection to the server
2. Transmit the request-headers
3. Initialize a variable R to the estimated round-trip time to the server (e.g., based on the time it took to establish the connection), or to a constant value of 5 seconds if the round-trip time is not available.
4. Compute T = R * (2**N), where N is the number of previous retries of this request.
5. Wait either for an error response from the server, or for T seconds (whichever comes first)
6. If no error response is received, after T seconds transmit the body of the request.
7. If client sees that the connection is closed prematurely, repeat from step 1 until the request is accepted, an error response is received, or the user becomes impatient and terminates the retry process.

If at any point an error status is received, the client

- SHOULD NOT continue and
- SHOULD close the connection if it has not completed sending the request message.

9 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 14.23) MUST accompany all HTTP/1.1 requests.

9.1 Safe and Idempotent Methods

9.1.1 Safe Methods

Implementors 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 might 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 NOT have the significance of taking an action other than retrieval. These methods ought to 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.

9.1.2 Idempotent Methods

Methods can also have the property of “idempotence” in that (aside from error or expiration issues) the side-effects of N > 0 identical requests is the same as for a single request. The methods GET, HEAD, PUT and DELETE share this property. Also, the methods OPTIONS and TRACE SHOULD NOT have side effects, and so are inherently idempotent.

However, it is possible that a sequence of several requests is non-idempotent, even if all of the methods executed in that sequence are idempotent. (A sequence is idempotent if a single execution of the entire sequence always yields a result that is not changed by a reexecution of all, or part, of that sequence.) For example, a sequence is non-idempotent if its result depends on a value that is later modified in the same sequence.

A sequence that never has side effects is idempotent, by definition (provided that no concurrent operations are being executed on the same set of resources).

9.2 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.

Responses to this method are not cacheable.

If the OPTIONS request includes an entity-body (as indicated by the presence of Content-Length or Transfer-Encoding), then the media type MUST be indicated by a Content-Type field. Although this specification does not define any use for such a body, future extensions to HTTP might use the OPTIONS body to make more detailed queries on the server. A server that does not support such an extension MAY discard the request body.

If the Request-URI is an asterisk (“*”), the OPTIONS request is intended to apply to the server in general rather than to a specific resource. Since a server’s communication options typically depend on the resource, the “*” request is only useful as a “ping” or “no-op” type of method; it does nothing beyond allowing the client to test the capabilities of the server. For example, this can be used to test a proxy for HTTP/1.1 compliance (or lack thereof).

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 that indicate optional features implemented by the server and applicable to that resource (e.g., Allow), possibly including extensions not defined by this specification. The response body, if any, SHOULD also include information about the communication options. The format for such a body is not defined by this specification, but might be defined by future extensions to HTTP. Content negotiation MAY be used to select the appropriate response format. If no response body is included, the response MUST include a Content-Length field with a field-value of “0”.

The Max-Forwards request-header field MAY be used to target a specific proxy in the request chain. When a proxy receives an OPTIONS request on an absoluteURI for which request forwarding is permitted, the proxy MUST check for a Max-Forwards field. If the Max-Forwards field-value is zero (“0”), the proxy MUST NOT forward the message; instead, the proxy SHOULD respond with its own communication options. If the Max-Forwards field-value is an integer greater than zero, the proxy MUST decrement the field-value when it forwards the request. If no Max-Forwards field is present in the request, then the forwarded request MUST NOT include a Max-Forwards field.

9.3 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, If-Unmodified-Since, If-Match, If-None-Match, or If-Range header field. A conditional GET method requests that the entity be transferred only under the circumstances described by the conditional header field(s). 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 entity be transferred, as described in section 14.35. 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 is cacheable if and only if it meets the requirements for HTTP caching described in section 13.

See section 15.1.3 for security considerations when used for forms.

9.4 HEAD

The HEAD method is identical to GET except that the server MUST NOT return a message-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 entity implied by the request 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 cacheable 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 entity (as would be indicated by a change in Content-Length, Content-MD5, ETag or Last-Modified), then the cache MUST treat the cache entry as stale.

9.5 POST

The POST method is used to request that the origin 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, 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.

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 which describes the status of the request and refers to the new resource, and a Location header (see section 14.30).

Responses to this method are not cacheable, unless the response includes appropriate Cache-Control or Expires header fields. However, the 303 (See Other) response can be used to direct the user agent to retrieve a cacheable resource.

POST requests MUST obey the message transmission requirements set out in section 8.2.

See section 15.1.3 for security considerations.

9.6 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. The recipient of the entity MUST NOT ignore any Content-* (e.g. Content-Range) headers that it does not understand or implement and MUST return a 501 (Not Implemented) response in such cases.

If the request passes through a cache and the Request-URI identifies one or more currently cached entities, those entries SHOULD be treated as stale. Responses to this method are not cacheable.

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. That resource might 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 might 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 might result in several other URIs being defined by the origin server.

HTTP/1.1 does not define how a PUT method affects the state of an origin server.

PUT requests MUST obey the message transmission requirements set out in section 8.2.

Unless otherwise specified for a particular entity-header, the entity-headers in the PUT request SHOULD be applied to the resource created or modified by the PUT.

9.7 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 action has been enacted but the response does not include an entity.

If the request passes through a cache and the `Request-URI` identifies one or more currently cached entities, those entries SHOULD be treated as stale. Responses to this method are not cacheable.

9.8 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 14.31). A TRACE request MUST NOT include an entity.

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 14.45) 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 the request is valid, the response SHOULD contain the entire request message in the entity-body, with a `Content-Type` of “message/http”. Responses to this method MUST NOT be cached.

9.9 CONNECT

This specification reserves the method name CONNECT for use with a proxy that can dynamically switch to being a tunnel (e.g. SSL tunneling [44]).

10 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.

10.1 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. There are no required headers for this class of status code. 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.

A client MUST be prepared to accept one or more 1xx status responses prior to a regular response, even if the client does not expect a 100 (Continue) status message. Unexpected 1xx status responses MAY be ignored by a user agent.

Proxies MUST forward 1xx responses, unless the connection between the proxy and its client has been closed, or unless the proxy itself requested the generation of the 1xx response. (For example, if a proxy adds a “Expect: 100-continue” field when it forwards a request, then it need not forward the corresponding 100 (Continue) response(s).)

10.1.1 100 Continue

The client SHOULD 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. See section 8.2.3 for detailed discussion of the use and handling of this status code.

10.1.2 101 Switching Protocols

The server understands and is willing to comply with the client's request, via the `Upgrade` message header field (section 14.42), 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 be switched only 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 might be advantageous when delivering resources that use such features.

10.2 Successful 2xx

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

10.2.1 200 OK

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

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

`HEAD` the entity-header fields corresponding to the requested resource are sent in the response without any message-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.

10.2.2 201 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 URI for the resource given by a `Location` header field. The response SHOULD include an entity containing a list of resource characteristics and location(s) from which the user or user agent can choose the one most appropriate. The entity format is specified by the media type given in the `Content-Type` header field. The origin server MUST create the resource before returning the 201 status code. If the action cannot be carried out immediately, the server SHOULD respond with 202 (Accepted) response instead.

A 201 response MAY contain an `ETag` response header field indicating the current value of the entity tag for the requested variant just created, see section 14.19.

10.2.3 202 Accepted

The request has been accepted for processing, but the processing has not been completed. The request might or might not eventually be acted upon, as it might be disallowed when processing actually takes place. There is no facility 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.

10.2.4 203 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 might 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).

10.2.5 204 No Content

The server has fulfilled the request but does not need to return an entity-body, and might want to return updated metainformation. The response MAY include new or updated metainformation in the form of entity-headers, which if present SHOULD be associated with the requested variant.

If the client is a user agent, it SHOULD NOT change its document view from that which caused the request to be sent. 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, although any new or updated metainformation SHOULD be applied to the document currently in the user agent's active view.

The 204 response MUST NOT include a message-body, and thus is always terminated by the first empty line after the header fields.

10.2.6 205 Reset Content

The server has fulfilled the request and the user agent SHOULD reset the document view which caused the request to be sent. 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 NOT include an entity.

10.2.7 206 Partial Content

The server has fulfilled the partial `GET` request for the resource. The request MUST have included a `Range` header field (section 14.35) indicating the desired range, and MAY have included an `If-Range` header field (section 14.27) to make the request conditional.

The response MUST include the following header fields:

- Either a `Content-Range` header field (section 14.16) indicating the range included with this response, or a multipart/byteranges `Content-Type` including `Content-Range` fields for each part. If a `Content-Length` header field is present in the response, its value MUST match the actual number of OCTETs transmitted in the message-body.
- `Date`
- `ETag` and/or `Content-Location`, if the header would have been sent in a 200 response to the same request
- `Expires`, `Cache-Control`, and/or `Vary`, if the field-value might differ from that sent in any previous response for the same variant

If the 206 response is the result of an `If-Range` request that used a strong cache validator (see section 13.3.3), the response SHOULD NOT include other entity-headers. If the response is the result of an `If-Range` request that used a weak validator, the response MUST NOT include other entity-headers; this prevents inconsistencies between cached entity-bodies and updated headers. Otherwise, the response MUST include all of the entity-headers that would have been returned with a 200 (OK) response to the same request.

A cache MUST NOT combine a 206 response with other previously cached content if the `ETag` or `Last-Modified` headers do not match exactly, see 13.5.4.

A cache that does not support the Range and Content-Range headers MUST NOT cache 206 (Partial) responses.

10.3 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 client SHOULD detect infinite redirection loops, since such loops generate network traffic for each redirection.

> Note: previous versions of this specification recommended a maximum of five redirections. Content developers should be aware that there might be clients that implement such a fixed limitation.

10.3.1 300 Multiple Choices

The requested resource corresponds to any one of a set of representations, each with its own specific location, and agent-driven negotiation information (section 12) is being provided so that the user (or user agent) can select a preferred representation and redirect its request to that location.

Unless it was a HEAD request, the response SHOULD include an entity containing a list of resource characteristics and location(s) from which the user or user agent can choose the one most appropriate. The entity format is specified by the media type given in the Content-Type header field. Depending upon the format and the capabilities of the user agent, selection of the most appropriate choice MAY be performed automatically. However, this specification does not define any standard for such automatic selection.

If the server has a preferred choice of representation, it SHOULD include the specific URI for that representation in the Location field; user agents MAY use the Location field value for automatic redirection. This response is cacheable unless indicated otherwise.

10.3.2 301 Moved Permanently

The requested resource has been assigned a new permanent URI and any future references to this resource SHOULD use one of the returned URIs. Clients with link editing capabilities ought to 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 cacheable unless indicated otherwise.

The new permanent URI SHOULD be given by the Location field in the response. Unless the request method was HEAD, the entity 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.

10.3.3 302 Found

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

The temporary URI SHOULD be given by the Location field in the response. Unless the request method was HEAD, the entity 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.

> Note: RFC 1945 and RFC 2068 specify that the client is not allowed to change the method on the redirected request. However, most existing user agent implementations treat 302 as if it were a 303 response, performing a GET on the `Location` field-value regardless of the original request method. The status codes 303 and 307 have been added for servers that wish to make unambiguously clear which kind of reaction is expected of the client.

10.3.4 303 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 URI is not a substitute reference for the originally requested resource. The 303 response MUST NOT be cached, but the response to the second (redirected) request might be cacheable.

The different URI SHOULD be given by the `Location` field in the response. Unless the request method was HEAD, the entity of the response SHOULD contain a short hypertext note with a hyperlink to the new URI(s).

> Note: Many pre-HTTP/1.1 user agents do not understand the 303 status. When interoperability with such clients is a concern, the 302 status code may be used instead, since most user agents react to a 302 response as described here for 303.

10.3.5 304 Not Modified

If the client has performed a conditional GET request and access is allowed, but the document has not been modified, the server SHOULD respond with this status code. The 304 response MUST NOT contain a message-body, and thus is always terminated by the first empty line after the header fields.

The response MUST include the following header fields:

- `Date`, unless its omission is required by section 14.18.1

  If a clockless origin server obeys these rules, and proxies and clients add their own `Date` to any response received without one (as already specified by [RFC 2068], section 14.19), caches will operate correctly.

- `ETag` and/or `Content-Location`, if the header would have been sent in a 200 response to the same request
- `Expires`, `Cache-Control`, and/or `Vary`, if the field-value might differ from that sent in any previous response for the same variant

If the conditional GET used a strong cache validator (see section 13.3.3), the response SHOULD NOT include other entity-headers. Otherwise (i.e., the conditional GET used a weak validator), the response MUST NOT include other entity-headers; this prevents inconsistencies between cached entity-bodies and updated headers.

If a 304 response indicates an entity not currently cached, then the cache MUST disregard the response and repeat the request without the conditional.

If a cache uses a received 304 response to update a cache entry, the cache MUST update the entry to reflect any new field values given in the response.

10.3.6 305 Use Proxy

The requested resource MUST be accessed through the proxy given by the `Location` field. The `Location` field gives the URI of the proxy. The recipient is expected to repeat this single request via the proxy. 305 responses MUST only be generated by origin servers.

> Note: RFC 2068 was not clear that 305 was intended to redirect a single request, and to be generated by origin servers only. Not observing these limitations has significant security consequences.

10.3.7 306 (Unused)

The 306 status code was used in a previous version of the specification, is no longer used, and the code is reserved.

10.3.8 307 Temporary Redirect

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 cacheable if indicated by a `Cache-Control` or `Expires` header field.

The temporary URI SHOULD be given by the `Location` field in the response. Unless the request method was HEAD, the entity of the response SHOULD contain a short hypertext note with a hyperlink to the new URI(s) , since many pre-HTTP/1.1 user agents do not understand the 307 status. Therefore, the note SHOULD contain the information necessary for a user to repeat the original request on the new URI.

If the 307 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.

10.4 Client Error 4xx

The 4xx class of status code is intended for cases in which the client seems to have erred. 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. User agents SHOULD display any included entity to the user.

If the client is sending data, a server implementation using TCP SHOULD be careful to ensure that the client acknowledges receipt of the packet(s) containing the response, before the server closes the input connection. If the client continues sending data to the server after the close, the server's TCP stack 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.

10.4.1 400 Bad Request

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

10.4.2 401 Unauthorized

The request requires user authentication. The response MUST include a `WWW-Authenticate` header field (section 14.47) containing a `challenge` applicable to the requested resource. The client MAY repeat the request with a suitable `Authorization` header field (section 14.8). 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 might include relevant diagnostic information. HTTP access authentication is explained in "HTTP Authentication: Basic and Digest Access Authentication" [43].

10.4.3 402 Payment Required

This code is reserved for future use.

10.4.4 403 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. If the server does not wish to make this information available to the client, the status code 404 (Not Found) can be used instead.

10.4.5 404 Not Found

The server has not found anything matching the `Request-URI`. No indication is given of whether the condition is temporary or permanent. 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. 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.

10.4.6 405 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.

10.4.7 406 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.

Unless it was a HEAD request, the response SHOULD include an entity containing a list of available entity characteristics and location(s) from which the user or user agent can choose the one most appropriate. The entity format is specified by the media type given in the `Content-Type` header field. Depending upon the format and the capabilities of the user agent, selection of the most appropriate choice MAY be performed automatically. However, this specification does not define any standard for such automatic selection.

> Note: 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 to 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 could be unacceptable, a user agent SHOULD temporarily stop receipt of more data and query the user for a decision on further actions.

10.4.8 407 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 14.33) 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 14.34). HTTP access authentication is explained in “HTTP Authentication: Basic and Digest Access Authentication” [43].

10.4.9 408 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.

10.4.10 409 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 might 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 might not be possible and is not required.

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

10.4.11 410 Gone

The requested resource is no longer available at the server and no forwarding address is known. This condition is expected to 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 cacheable 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.

10.4.12 411 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 message-body in the request message.

10.4.13 412 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.

10.4.14 413 Request Entity Too Large

The server is refusing to process a request because the request entity is larger than the server is willing or able to process. The server MAY close the connection to prevent the client from continuing the request.

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

10.4.15 414 Request-URI Too Long

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

10.4.16 415 Unsupported Media Type

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

10.4.17 416 Requested Range Not Satisfiable

A server SHOULD return a response with this status code if a request included a `Range` request-header field (section 14.35) , and none of the range-specifier values in this field overlap the current extent of the selected resource, and the request did not include an `If-Range` request-header field. (For byte-ranges, this means that the first-byte-pos of all of the byte-range-spec values were greater than the current length of the selected resource.)

When this status code is returned for a byte-range request, the response SHOULD include a `Content-Range` entity-header field specifying the current length of the selected resource (see section 14.16). This response MUST NOT use the `multipart/byteranges` content-type.

10.4.18 417 Expectation Failed

The expectation given in an `Expect` request-header field (see section 14.20) could not be met by this server, or, if the server is a proxy, the server has unambiguous evidence that the request could not be met by the next-hop server.

10.5 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. 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. User agents SHOULD display any included entity to the user. These response codes are applicable to any request method.

10.5.1 500 Internal Server Error

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

10.5.2 501 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.

10.5.3 502 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.

10.5.4 503 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.

10.5.5 504 Gateway Timeout

The server, while acting as a gateway or proxy, did not receive a timely response from the upstream server specified by the URI (e.g. HTTP, FTP, LDAP) or some other auxiliary server (e.g. DNS) it needed to access in attempting to complete the request.

> Note: Note to implementors: some deployed proxies are known to return 400 or 500 when DNS lookups time out.

10.5.6 505 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.

11 Access Authentication

HTTP provides several OPTIONAL challenge-response authentication mechanisms which can be used by a server to challenge a client request and by a client to provide authentication information. The general framework for access authentication, and the specification of "basic" and "digest" authentication, are specified in "HTTP Authentication: Basic and Digest Access Authentication" [43]. This specification adopts the definitions of "`challenge`" and "`credentials`" from that specification.

12 Content Negotiation

Most HTTP responses include an entity which contains information for interpretation by a human user. Naturally, it is desirable to supply the user with the "best available" entity corresponding to the request. Unfortunately for servers and caches, not all users have the same preferences for what is "best," and not all user agents are equally capable of rendering all entity types. For that reason, HTTP has provisions for several mechanisms for "content negotiation" -- the process of selecting the best representation for a given response when there are multiple representations available.

> Note: This is not called "format negotiation" because the alternate representations may be of the same media type, but use different capabilities of that type, be in different languages, etc.

Any response containing an entity-body MAY be subject to negotiation, including error responses.

There are two kinds of content negotiation which are possible in HTTP: server-driven and agent-driven negotiation. These two kinds of negotiation are orthogonal and thus may be used separately or in combination. One method of combination, referred to as transparent negotiation, occurs when a cache uses the agent-driven negotiation information provided by the origin server in order to provide server-driven negotiation for subsequent requests.

12.1 Server-driven Negotiation

If the selection of the best representation for a response is made by an algorithm located at the server, it is called server-driven negotiation. Selection is based on the available representations of the response (the dimensions over which it can vary; e.g. language, content-coding, etc.) and the contents of particular header fields in the request message or on other information pertaining to the request (such as the network address of the client).

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

Server-driven negotiation has disadvantages:

1. It is impossible for the server to accurately determine what might be "best" for any given user, since that would require complete knowledge of both the capabilities of the user agent and the intended use for the response (e.g., does the user want to view it on screen or print it on paper?).

2. Having the user agent describe its capabilities in every request can be both very inefficient (given that only a small percentage of responses have multiple representations) and a potential violation of the user's privacy.

3. It complicates the implementation of an origin server and the algorithms for generating responses to a request.

4. It may limit a public cache's ability to use the same response for multiple user's requests.

HTTP/1.1 includes the following request-header fields for enabling server-driven negotiation through description of user agent capabilities and user preferences: `Accept` (section 14.1), `Accept-Charset` (section 14.2), `Accept-`

Encoding (section 14.3), Accept-Language (section 14.4), and User-Agent (section 14.43). However, an origin server is not limited to these dimensions and MAY vary the response based on any aspect of the request, including information outside the request-header fields or within extension header fields not defined by this specification.

The Vary header field can be used to express the parameters the server uses to select a representation that is subject to server-driven negotiation. See section 13.6 for use of the Vary header field by caches and section 14.44 for use of the Vary header field by servers.

12.2 Agent-driven Negotiation

With agent-driven negotiation, selection of the best representation for a response is performed by the user agent after receiving an initial response from the origin server. Selection is based on a list of the available representations of the response included within the header fields or entity-body of the initial response, with each representation identified by its own URI. Selection from among the representations may be performed automatically (if the user agent is capable of doing so) or manually by the user selecting from a generated (possibly hypertext) menu.

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

Agent-driven negotiation suffers from the disadvantage of needing a second request to obtain the best alternate representation. This second request is only efficient when caching is used. In addition, this specification does not define any mechanism for supporting automatic selection, though it also does not prevent any such mechanism from being developed as an extension and used within HTTP/1.1.

HTTP/1.1 defines the 300 (Multiple Choices) and 406 (Not Acceptable) status codes for enabling agent-driven negotiation when the server is unwilling or unable to provide a varying response using server-driven negotiation.

12.3 Transparent Negotiation

Transparent negotiation is a combination of both server-driven and agent-driven negotiation. When a cache is supplied with a form of the list of available representations of the response (as in agent-driven negotiation) and the dimensions of variance are completely understood by the cache, then the cache becomes capable of performing server-driven negotiation on behalf of the origin server for subsequent requests on that resource.

Transparent negotiation has the advantage of distributing the negotiation work that would otherwise be required of the origin server and also removing the second request delay of agent-driven negotiation when the cache is able to correctly guess the right response.

This specification does not define any mechanism for transparent negotiation, though it also does not prevent any such mechanism from being developed as an extension that could be used within HTTP/1.1.

13 Caching in HTTP

HTTP is typically used for distributed information systems, where performance can be improved by the use of response caches. 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.

Caching would be useless if it did not significantly improve performance. 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).

Requirements for performance, availability, and disconnected operation require us to be able to relax the goal of semantic transparency. 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 might be incompatible with certain server applications (such as those for ordering merchandise), the protocol requires that transparency be relaxed

- only by an explicit protocol-level request when relaxed by client or origin server
- only with an explicit warning to 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 required by all parties.
2. Protocol features that allow an origin server or user agent to explicitly request and control non-transparent operation.
3. Protocol features that allow a cache to attach warnings to responses that do not preserve the requested approximation of semantic transparency.

A basic principle is that it must be possible for the clients to detect any potential relaxation of semantic transparency.

> Note: The server, cache, or client implementor might be faced with design decisions not explicitly discussed in this specification. If a decision might affect semantic transparency, the implementor ought to err on the side of maintaining transparency unless a careful and complete analysis shows significant benefits in breaking transparency.

13.1.1 Cache Correctness

A correct cache MUST respond to a request with the most up-to-date response held by the cache that is appropriate to the request (see sections 13.2.5, 13.2.6, and 13.12) which meets one of the following conditions:

1. It has been checked for equivalence with what the origin server would have returned by revalidating the response with the origin server (section 13.3);
2. It is “fresh enough” (see section 13.2). In the default case, this means it meets the least restrictive freshness requirement of the client, origin server, and cache (see section 14.9); if the origin server so specifies, it is the freshness requirement of the origin server alone.

   If a stored response is not “fresh enough” by the most restrictive freshness requirement of both the client and the origin server, in carefully considered circumstances the cache MAY still return the response with the appropriate `Warning` header (see section 13.1.5 and 14.46), unless such a response is prohibited (e.g., by a “`no-store`” cache-directive, or by a “`no-cache`” cache-request-directive; see section 14.9).
3. It is an appropriate 304 (Not Modified), 305 (Proxy Redirect), or error (4xx or 5xx) response message.

If the cache can not communicate with the origin server, then a correct cache SHOULD respond as above if the response can be correctly served from the cache; if not it MUST return an error or warning indicating that there was a communication failure.

If a cache receives a response (either an entire response, or a 304 (Not Modified) response) that it would normally forward to the requesting client, and the received response is no longer fresh, the cache SHOULD forward it to the requesting client without adding a new `Warning` (but without removing any existing `Warning` headers). A cache SHOULD NOT attempt to revalidate a response simply because that response became stale in transit; this might lead to an infinite loop. A user agent that receives a stale response without a `Warning` MAY display a warning indication to the user.

13.1.2 Warnings

Whenever a cache returns a response that is neither first-hand nor "fresh enough" (in the sense of condition 2 in section 13.1.1), it MUST attach a warning to that effect, using a `Warning` general-header. The `Warning` header and the currently defined warnings are described in section 14.46. The warning allows clients 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 assigned three digit `warn-codes`. The first digit indicates whether the `Warning` MUST or MUST NOT be deleted from a stored cache entry after a successful revalidation:

1xx Warnings that describe the freshness or revalidation status of the response, and so MUST be deleted after a successful revalidation. 1XX `warn-codes` MAY be generated by a cache only when validating a cached entry. It MUST NOT be generated by clients.

2xx Warnings that describe some aspect of the entity body or entity headers that is not rectified by a revalidation (for example, a lossy compression of the entity bodies) and which MUST NOT be deleted after a successful revalidation.

See section 14.46 for the definitions of the codes themselves.

HTTP/1.0 caches will cache all `Warnings` in responses, without deleting the ones in the first category. Warnings in responses that are passed to HTTP/1.0 caches carry an extra `warning-date` field, which prevents a future HTTP/1.1 recipient from believing an erroneously cached `Warning`.

Warnings also carry a warning text. The text MAY be in any appropriate natural language (perhaps based on the client's `Accept` headers), and include an OPTIONAL indication of what character set is used.

Multiple warnings 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 might provide the same warning with texts in both English and Basque.

When multiple warnings are attached to a response, it might 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.

13.1.3 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 might 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 is 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 the approximation of semantic transparency (for example, "`max-stale`" or "`public`").

The cache-control directives are described in detail in section 14.9.

13.1.4 Explicit User Agent Warnings

Many user agents make it possible for users to override the basic caching mechanisms. For example, the user agent might 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`" to every request. The user agent SHOULD NOT default to either non-transparent behavior, or behavior that results in abnormally ineffective caching, but MAY be explicitly configured to do so by an explicit action of the user.

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 entity 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 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 indicate this state to the user (for example, by a display of a picture of currency in flames) so that the user does not inadvertently consume excess resources or suffer from excessive latency.

13.1.5 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 ought 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) enabling the client software to alert the user that there might be a potential problem.

It also allows the user agent to take steps to obtain a first-hand or fresh response. For this reason, a cache SHOULD NOT return a stale response if the client explicitly requests a first-hand or fresh one, unless it is impossible to comply for technical or policy reasons.

13.1.6 Client-controlled Behavior

While the origin server (and to a lesser extent, intermediate caches, by their contribution to the age of a response) are the primary source of expiration information, in some cases the client might 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 of 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 further relax the cache's approximation of 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 might violate the origin server's specified constraints on semantic transparency, but might be necessary to support disconnected operation, or high availability in the face of poor connectivity.

13.2 Expiration Model

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.

The expiration mechanism applies only to responses taken from a cache and not to first-hand responses forwarded immediately to the requesting client.

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. See section 14.9.4 for a more restrictive way to force revalidation.

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 14.9).

Servers specify explicit expiration times using either the `Expires` header, or the `max-age` directive of the `Cache-Control` header.

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. See section 13.13 for an explanation of the difference between caches and history mechanisms.

13.2.2 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 might compromise semantic transparency, they ought to used cautiously, and we encourage origin servers to provide explicit expiration times as much as possible.

13.2.3 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 13.2.4; 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." Hosts that use HTTP, but especially hosts running origin servers and caches, SHOULD use NTP [28] or some similar protocol to synchronize their clocks to a globally accurate time standard.

HTTP/1.1 requires origin servers to send a `Date` header, if possible, with every response, giving the time at which the response was generated (see section 14.18). We use the term "date_value" to denote the value of the `Date` header, in a form appropriate for arithmetic operations.

HTTP/1.1 uses the `Age` response-header to convey the estimated age of the response message when obtained from a cache. The `Age` field value is the cache's estimate of the amount of time since the response was generated or revalidated by the origin server.

In essence, the `Age` value is the sum of the time that the response has been resident in each of the caches along the path from the origin server, plus the amount of time it has been in transit along network paths.

We use the term "age_value" to denote the value of the `Age` header, in a form appropriate for arithmetic operations.

A response's age can be calculated in two entirely independent ways:

1. now minus date_value, if the local clock is reasonably well synchronized to the origin server's clock. If the result is negative, the result is replaced by zero.

2. 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.

Because of network-imposed delays, some significant interval might pass between 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, response_time - date_value);
corrected_received_age = max(apparent_age, age_value);
response_delay = response_time - request_time;
corrected_initial_age = corrected_received_age + response_delay;
resident_time = now - response_time;
current_age   = corrected_initial_age + resident_time;
```

The `current_age` of a cache entry is calculated by adding the amount of time (in seconds) since the cache entry was last validated by the origin server to the `corrected_initial_age`. When a response is generated from a cache entry, the cache MUST include a single `Age` header field in the response with a value equal to the cache entry's `current_age`.

The presence of an `Age` header field in a response implies that a response is not first-hand. However, the converse is not true, since the lack of an `Age` header field in a response does not imply that the response is first-hand unless all caches along the request path are compliant with HTTP/1.1 (i.e., older HTTP caches did not implement the `Age` header field).

13.2.4 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 13.2.3; this section describes how to calculate the freshness lifetime, and to determine if a response has expired. In the discussion below, the values can be represented in any form appropriate for arithmetic operations.

We use the term "`expires_value`" to denote the value of the `Expires` header. We use the term "`max_age_value`" to denote an appropriate value of the number of seconds carried by the "`max-age`" directive of the `Cache-Control` header in a response (see section 14.9.3).

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 none of `Expires`, `Cache-Control: max-age`, or `Cache-Control: s-maxage` (see section 14.9.3) appears in the response, and the response does not include other restrictions on caching, the cache MAY compute a freshness lifetime using a heuristic. The cache MUST attach `Warning 113` to any response whose age is more than 24 hours if such warning has not already been added.

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)
```

13.2.5 Disambiguating Expiration Values

Because expiration values are assigned optimistically, it is possible for two caches to contain fresh values for the same resource that are different.

If a client performing a retrieval receives a non-first-hand response for a request 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 14.9), to force a check with the origin server.

If a cache has two fresh responses for the same representation with different validators, it MUST use the one with the more recent `Date` header. This situation might arise because the cache is pooling responses from other caches, or because a client has asked for a reload or a revalidation of an apparently fresh cache entry.

13.2.6 Disambiguating Multiple Responses

Because a client might 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 might receive responses in an order different from that in which the origin server sent them. We would like the client to use the most recently generated response, even if older responses are still apparently fresh.

Neither the entity tag nor the expiration value can impose an ordering on responses, since it is possible that a later response intentionally carries an earlier expiration time. The `Date` values are ordered to a granularity of one second.

When a client tries to revalidate a cache entry, and the response it receives contains a `Date` header that appears to be older than the one for the existing entry, 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.

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.

13.3 Validation Model

When a cache has a stale entry 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 entry 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 entry is good, and we do not want to pay the overhead of an extra round trip if the cached entry 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 (user agent or proxy 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 entity, and, if they match (see section 13.3.3), 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.

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 a validator matches or if and only if no validators match.

> Note: a response that lacks a 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 validator for the entity, which means it will not be refreshable after it expires.

13.3.1 Last-Modified Dates

The `Last-Modified` entity-header field value is often used as a cache validator. In simple terms, a cache entry is considered to be valid if the entity has not been modified since the `Last-Modified` value.

13.3.2 Entity Tag Cache Validators

The `ETag` response-header field value, an entity tag, provides for an “opaque” cache validator. This might allow more reliable validation in situations where it is inconvenient to store modification dates, where the one-second resolution of HTTP date values is not sufficient, or where the origin server wishes to avoid certain paradoxes that might arise from the use of modification dates.

Entity Tags are described in section 3.11. The headers used with entity tags are described in sections 14.19, 14.24, 14.26 and 14.44.

13.3.3 Weak and Strong Validators

Since both origin servers and caches will compare two validators to decide if they represent the same or different entities, one normally would expect that if the entity (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 might be cases when a server prefers to change the validator only on semantically significant changes, and not when insignificant aspects of the entity change. A validator that does not always change when the resource changes is a “weak validator.”

Entity tags are normally “strong validators,” but the protocol provides a mechanism to tag an entity tag as “weak.” 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 entity, while a weak validator is part of an identifier for a set of semantically equivalent entities.

> 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 might be modified twice during a single second.

Support for weak validators is optional. However, weak validators allow for more efficient caching of equivalent objects; for example, a hit counter on a site is probably good enough if it is updated every few days or weeks, and any value during that period is likely "good enough" to be equivalent.

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 might end up with an internally inconsistent entity.

Clients MAY issue simple (non-subrange) GET requests with either weak validators or strong validators. Clients MUST NOT use weak validators in other forms of request.

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 both MUST NOT 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.

An entity tag is strong unless it is explicitly tagged as weak. Section 3.11 gives the syntax for entity tags.

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 includes a `Date` value, which gives the time when the origin server sent 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 includes a `Date` value, which gives the time when the origin server sent 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 sent by the origin server during the same second, but both had the same `Last-Modified` time, then at least one of those responses would have a `Date` value equal to its `Last-Modified` time. The arbitrary 60-second limit guards against the possibility that the `Date` and `Last-Modified` values are generated from different clocks, or at somewhat different times during the preparation of the response. An implementation MAY use a value larger than 60 seconds, if it is believed that 60 seconds is too short.

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 conditional request, other than a full-body GET request, MUST use the strong comparison function to evaluate the condition.

These rules allow 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 Entity Tags and Last-Modified Dates

We adopt a set of rules and recommendations for origin servers, clients, and caches regarding when various validator types ought to be used, and for what purposes.

HTTP/1.1 origin servers:

- SHOULD send an entity tag validator unless it is not feasible to generate one.
- MAY send a weak entity tag instead of a strong entity tag, if performance considerations support the use of weak entity tags, or if it is unfeasible to send a strong entity tag.
- 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 entity tag and a `Last-Modified` value.

In order to be legal, a strong entity tag MUST change whenever the associated entity value changes in any way. A weak entity tag SHOULD change whenever the associated entity changes in a semantically significant way.

> Note: in order to provide semantically transparent caching, an origin server must avoid reusing a specific strong entity tag value for two different entities, or reusing a specific weak entity tag value for two semantically different entities. Cache entries might persist for arbitrarily long periods, regardless of expiration times, so it might 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 entity tag has been provided by the origin server, MUST use that entity tag in any cache-conditional request (using `If-Match` or `If-None-Match`).
- 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 entity tag 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 origin server, upon receiving a conditional request that includes both a Last-Modified date (e.g., in an `If-Modified-Since` or `If-Unmodified-Since` header field) and one or more entity tags (e.g., in an `If-Match`, `If-None-Match`, or `If-Range` header field) as cache validators, MUST NOT return a response status of 304 (Not Modified) unless doing so is consistent with all of the conditional header fields in the request.

An HTTP/1.1 caching proxy, upon receiving a conditional request that includes both a Last-Modified date and one or more entity tags as cache validators, MUST NOT return a locally cached response to the client unless that cached response is consistent with all of the conditional header fields in the request.

Note: 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 entity tags. 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 Non-validating Conditionals

The principle behind entity tags 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.

13.4 Response Cacheability

Unless specifically constrained by a cache-control (section 14.9) directive, a caching system MAY always store a successful response (see section 13.8) 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). A client can usually detect that such a response was taken from a cache by comparing the `Date` header to the current time.

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

However, in some cases it might be inappropriate for a cache to retain an entity, or to return it in response to a subsequent request. This might 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 resource entities, or portions thereof, are not to be cached regardless of other considerations.

Note that section 14.8 normally prevents a shared cache from saving and returning a response to a previous request if that request included an `Authorization` header.

A response received with a status code of 200, 203, 206, 300, 301 or 410 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. However, a cache that does not support the `Range` and `Content-Range` headers MUST NOT cache 206 (Partial Content) responses.

A response received with any other status code (e.g. status codes 302 and 307) MUST NOT be returned in a reply to a subsequent request unless there are cache-control directives or another header(s) that explicitly allow it. For example, these include the following: an `Expires` header (section 14.21); a “`max-age`”, “`s-maxage`”, “`must-revalidate`”, “`proxy-revalidate`”, “`public`” or “`private`” cache-control directive (section 14.9).

13.5 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 might have to combine parts of a new response with what is held in the cache entry.

13.5.1 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 are 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 MUST be 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`
- `Proxy-Authenticate`
- `Proxy-Authorization`
- `TE`
- `Trailers`
- `Transfer-Encoding`
- `Upgrade`

All other headers defined by HTTP/1.1 are end-to-end headers.

Other hop-by-hop headers MUST be listed in a `Connection` header, (section 14.10) to be introduced into HTTP/1.1 (or later).

13.5.2 Non-modifiable Headers

Some features of the HTTP/1.1 protocol, such as Digest Authentication, depend on the value of certain end-to-end headers. A transparent proxy SHOULD NOT modify an end-to-end header unless the definition of that header requires or specifically allows that.

A transparent proxy MUST NOT modify any of the following fields in a request or response, and it MUST NOT add any of these fields if not already present:

- `Content-Location`
- `Content-MD5`
- `ETag`
- `Last-Modified`

A transparent proxy MUST NOT modify any of the following fields in a response:

- `Expires`

but it MAY add any of these fields if not already present. If an `Expires` header is added, it MUST be given a field-value identical to that of the `Date` header in that response.

A proxy MUST NOT modify or add any of the following fields in a message that contains the `no-transform` cache-control directive, or in any request:

- `Content-Encoding`
- `Content-Range`
- `Content-Type`

A non-transparent proxy MAY modify or add these fields to a message that does not include `no-transform`, but if it does so, it MUST add a Warning 214 (Transformation applied) if one does not already appear in the message (see section 14.46).

> Warning: unnecessary modification of end-to-end headers might 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.

The `Content-Length` field of a request or response is added or deleted according to the rules in section 4.4. A transparent proxy MUST preserve the entity-length (section 7.2.2) of the entity-body, although it MAY change the transfer-length (section 4.4).

13.5.3 Combining Headers

When a cache makes a validating request to a server, and the server provides a 304 (Not Modified) response or a 206 (Partial Content) response, the cache then constructs a response to send to the requesting client.

If the status code is 304 (Not Modified), the cache uses the entity-body stored in the cache entry as the entity-body of this outgoing response. If the status code is 206 (Partial Content) and the `ETag` or `Last-Modified` headers match exactly, the cache MAY combine the contents stored in the cache entry with the new contents received in the response and use the result as the entity-body of this outgoing response, (see 13.5.4).

The end-to-end headers stored in the cache entry are used for the constructed response, except that

- any stored `Warning` headers with warn-code 1xx (see section 14.46) MUST be deleted from the cache entry and the forwarded response.
- any stored `Warning` headers with warn-code 2xx MUST be retained in the cache entry and the forwarded response.
- any end-to-end headers provided in the 304 or 206 response MUST replace the corresponding headers from the cache entry.

Unless the cache decides to remove the cache entry, it MUST also replace the end-to-end headers stored with the cache entry with corresponding headers received in the incoming response, except for `Warning` headers as described immediately above. If a header field-name in the incoming response matches more than one header in the cache entry, all such old headers MUST be replaced.

In other words, the set of end-to-end headers received in the incoming response overrides all corresponding end-to-end headers stored with the cache entry (except for stored `Warning` headers with warn-code 1xx, which are deleted even if not overridden).

> Note: this rule allows an origin server to use a 304 (Not Modified) or a 206 (Partial Content) response to update any header associated with a previous response for the same entity or sub-ranges thereof, although it might not always be meaningful or correct to do so. This rule does not allow an origin server to use a 304 (Not Modified) or a 206 (Partial Content) response to entirely delete a header that it had provided with a previous response.

13.5.4 Combining Byte Ranges

A response might transfer only a subrange of the bytes of an entity-body, either because the request included one or more `Range` specifications, or because a connection was broken prematurely. After several such transfers, a cache might have received several ranges of the same entity-body.

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 have a cache validator.
- The two cache validators match using the strong comparison function (see section 13.3.3).

If either requirement is not met, 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.

13.6 Caching Negotiated Responses

Use of server-driven content negotiation (section 12.1), as indicated by the presence of a `Vary` header field in a response, alters the conditions and procedure by which a cache can use the response for subsequent requests. See section 14.44 for use of the `Vary` header field by servers.

A server SHOULD use the `Vary` header field to inform a cache of what request-header fields were used to select among multiple representations of a cacheable response subject to server-driven negotiation. The set of header fields named by the `Vary` field value is known as the “selecting” request-headers.

When the cache receives a subsequent request whose `Request-URI` specifies one or more cache entries including a `Vary` header field, the cache MUST NOT use such a cache entry to construct a response to the new request unless all of the selecting request-headers present in the new request match the corresponding stored request-headers in the original request.

The selecting request-headers from two requests are defined to match if and only if the selecting request-headers in the first request can be transformed to the selecting request-headers in the second request by adding or removing linear white space (LWS) at places where this is allowed by the corresponding BNF, and/or combining multiple message-header fields with the same field name following the rules about message headers in section 4.2.

A `Vary` header field-value of “*” always fails to match and subsequent requests on that resource can only be properly interpreted by the origin server.

If the selecting request header fields for the cached entry do not match the selecting request header fields of the new request, then the cache MUST NOT use a cached entry to satisfy the request unless it first relays the new request to the origin server in a conditional request and the server responds with 304 (Not Modified), including an entity tag or `Content-Location` that indicates the entity to be used.

If an entity tag was assigned to a cached representation, the forwarded request SHOULD be conditional and include the entity tags in an `If-None-Match` header field from all its cache entries for the resource. This conveys to the server the set of entities currently held by the cache, so that if any one of these entities matches the requested entity, the server can use the `ETag` header field in its 304 (Not Modified) response to tell the cache which entry is appropriate. If the entity-tag of the new response matches that of an existing entry, the new response SHOULD be used to update the header fields of the existing entry, and the result MUST be returned to the client.

If any of the existing cache entries contains only partial content for the associated entity, its entity-tag SHOULD NOT be included in the `If-None-Match` header field unless the request is for a range that would be fully satisfied by that entry.

If a cache receives a successful response whose `Content-Location` field matches that of an existing cache entry for the same `Request-URI`, whose entity-tag differs from that of the existing entry, and whose `Date` is more recent than that of the existing entry, the existing entry SHOULD NOT be returned in response to future requests and SHOULD be deleted from the cache.

13.7 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.

13.8 Errors or Incomplete Response Cache Behavior

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 13.5.4; 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).

If a cache receives a 5xx response while attempting to revalidate an entry, 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 unless the cached entry includes the "`must-revalidate`" cache-control directive (see section 14.9).

13.9 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 URIs 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 9.1.1 for related information.

13.10 Invalidation After Updates or Deletions

The effect of certain methods performed on a resource at the origin server might cause one or more existing cache entries to become non-transparently invalid. That is, although they might continue to be "fresh," they do not accurately reflect what the origin server would return for a new request on that resource.

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 might 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 will either remove all instances of that entity from its storage, or will 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 MUST cause a cache to invalidate an entity. This is either the entity referred to by the `Request-URI`, or by the `Location` or `Content-Location` 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`.

A cache that passes through requests for methods it does not understand SHOULD invalidate any entities referred to by the `Request-URI`.

13.11 Write-Through Mandatory

All methods that might 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. This does not prevent a proxy cache from sending a 100 (Continue) response before the inbound server has sent its final reply.

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.

13.12 Cache Replacement

If a new cacheable (see sections 14.9.2, 13.2.5, 13.2.6 and 13.8) response is received from a resource while any existing responses for the same resource are cached, the cache SHOULD use the new response to reply to the current request. It MAY insert it into cache storage and MAY, if it meets all other requirements, use it to respond to any future requests that would previously have caused the old response to be returned. If it inserts the new response into cache storage the rules in section 13.5.3 apply.

> Note: a new response that has an older `Date` header value than existing cached responses is not cacheable.

13.13 History Lists

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.

History mechanisms and caches are different. In particular history mechanisms SHOULD NOT try to show a semantically transparent view of the current state of a resource. Rather, a history mechanism is meant to show exactly what the user saw at the time when the resource was retrieved.

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.

This is not to be construed to prohibit the history mechanism from telling the user that a view might be stale.

> Note: if history list mechanisms unnecessarily prevent users from viewing stale resources, this will tend to force service authors to avoid using HTTP expiration controls and cache controls when they would otherwise like to. Service authors may consider it important that users not be presented with error messages or warning messages when they use navigation controls (such as BACK) to view previously fetched resources. Even though sometimes such resources ought not to cached, or ought to expire quickly, user interface considerations may force service authors to resort to other means of preventing caching (e.g. "once-only" URLs) in order not to suffer the effects of improperly functioning history mechanisms.

14 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.

14.1 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.

```
Accept         = "Accept" ":"
                 #( media-range [ accept-params ] )

media-range    = ( "*/*"
                 | ( type "/" "*" )
                 | ( type "/" subtype )
                 ) *( ";" parameter )
accept-params  = ";" "q" "=" qvalue *( accept-extension )
accept-extension = ";" token [ "=" ( token | quoted-string ) ]
```

The asterisk "*" character is used to group media types into ranges, with "*/*" indicating all media types and "type/*" indicating all subtypes of that type. The `media-range` MAY include media type parameters that are applicable to that range.

Each media-range MAY be followed by one or more `accept-params`, beginning with the "q" parameter for indicating a relative quality factor. The first "q" parameter (if any) separates the `media-range` parameter(s) from the `accept-params`. Quality factors allow the user or user agent to indicate the relative degree of preference for that media-range, using the qvalue scale from 0 to 1 (section 3.9). The default value is q=1.

> Note: Use of the "q" parameter name to separate media type parameters from `Accept` extension parameters is due to historical practice. Although this prevents any media type parameter named "q" from being used with a media range, such an event is believed to be unlikely given the lack of any "q" parameters in the IANA media type registry and the rare usage of any media type parameters in `Accept`. Future media types are discouraged from registering any parameter named "q".

The example

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

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 field is present, then it is assumed that the client accepts all media types. If an `Accept` header field is present, and if the server cannot send a response which is acceptable according to the combined `Accept` field value, then the server SHOULD send a 406 (not acceptable) response.

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,
        text/html;level=2;q=0.4, */*;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=2         = 0.4
text/html;level=3         = 0.7
```

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

14.2 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.

```
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
```

The special value "*", if present in the `Accept-Charset` field, matches every character set (including ISO-8859-1) which is not mentioned elsewhere in the `Accept-Charset` field. If no "*" is present in an `Accept-Charset` field, then all character sets not explicitly mentioned get a quality value of 0, except for ISO-8859-1, which gets a quality value of 1 if not explicitly mentioned.

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.

14.3 Accept-Encoding

The `Accept-Encoding` request-header field is similar to `Accept`, but restricts the content-codings (section 3.5) that are acceptable in the response.

```
    Accept-Encoding  = "Accept-Encoding" ":"
                       1#( codings [ ";" "q" "=" qvalue ] )
    codings          = ( content-coding | "*" )
Examples of its use are:
    Accept-Encoding: compress, gzip
    Accept-Encoding:
    Accept-Encoding: *
    Accept-Encoding: compress;q=0.5, gzip;q=1.0
    Accept-Encoding: gzip;q=1.0, identity; q=0.5, *;q=0
```

A server tests whether a content-coding is acceptable, according to an `Accept-Encoding` field, using these rules:

1. If the content-coding is one of the content-codings listed in the `Accept-Encoding` field, then it is acceptable, unless it is accompanied by a qvalue of 0. (As defined in section 3.9, a qvalue of 0 means "not acceptable.")

2. The special "*" symbol in an `Accept-Encoding` field matches any available content-coding not explicitly listed in the header field.

3. If multiple content-codings are acceptable, then the acceptable content-coding with the highest non-zero `qvalue` is preferred.

4. The "`identity`" content-coding is always acceptable, unless specifically refused because the `Accept-Encoding` field includes "`identity;q=0`", or because the field includes "`*;q=0`" and does not explicitly include the "`identity`" content-coding. If the `Accept-Encoding` field-value is empty, then only the "`identity`" encoding is acceptable.

If an `Accept-Encoding` field is present in a request, 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.

If no `Accept-Encoding` field is present in a request, the server MAY assume that the client will accept any content coding. In this case, if "`identity`" is one of the available content-codings, then the server SHOULD use the "`identity`" content-coding, unless it has additional information that a different content-coding is meaningful to the client.

> Note: If the request does not include an `Accept-Encoding` field, and if the "`identity`" content-coding is unavailable, then content-codings commonly understood by HTTP/1.0 clients (i.e., "`gzip`" and "`compress`") are preferred; some older clients improperly display messages sent with other content-codings. The server might also make this decision based on information about the particular user-agent or client.

> Note: Most HTTP/1.0 applications do not recognize or obey qvalues associated with content-codings. This means that qvalues will not work and are not permitted with `x-gzip` or `x-compress`.

14.4 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. Language tags are defined in section 3.10.

```
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 preference for the languages specified by that range. The quality value defaults to "q=1". For example,

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

would mean: "I prefer Danish, but will accept British English and other types of English." A `language-range` matches a `language-tag` if it exactly equals the tag, or if it exactly equals a prefix 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 range 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-tag`. 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.

It might 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 15.1.4.

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.

> Note: When making the choice of linguistic preference available to the user, we remind implementors of the fact that users are not familiar with the details of language matching as described above, and should provide appropriate guidance. As an example, users might assume that on selecting "en-gb", they will be served any kind of English document if British English is not available. A user agent might suggest in such a case to add "en" to get the best matching behavior.

14.5 Accept-Ranges

The `Accept-Ranges` response-header field allows the server to indicate its acceptance of range requests for a 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 resource involved. Range units are defined in section 3.12.

Servers that do not accept any kind of range request for a resource MAY send

```
Accept-Ranges: none
```

to advise the client not to attempt a range request.

14.6 Age

The `Age` response-header field conveys the sender's estimate of the amount of time since the response (or its revalidation) was generated at the origin server. A cached response is "fresh" if its age does not exceed its freshness lifetime. `Age` values are calculated as specified in section 13.2.3.

```
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 transmit an `Age` header with a value of 2147483648 (2^31). An HTTP/1.1 server that includes a cache MUST include an `Age` header field in every response generated from its own cache. Caches SHOULD use an arithmetic type of at least 31 bits of range.

14.7 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.

```
Allow   = "Allow" ":" #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 might have other means of communicating with the origin server.

14.8 Authorization

A user agent that wishes to authenticate itself with a server--usually, but not necessarily, after receiving a 401 response--does 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 “HTTP Authentication: Basic and Digest Access Authentication” [43]. If a request is authenticated and a `realm` specified, the same `credentials` SHOULD be valid for all other requests within this `realm` (assuming that the authentication scheme itself does not require otherwise, such as credentials that vary according to a challenge value or using synchronized clocks).

When a shared cache (see section 13.7) 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 “`s-maxage`” cache-control directive, the cache MAY use that response in replying to a subsequent request. But (if the specified maximum age has passed) 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. (This is the defined behavior for `s-maxage`.) If the response includes “`s-maxage=0`”, the proxy MUST always revalidate it before re-using it.

2. If the response includes the “`must-revalidate`” cache-control directive, the cache MAY use that response in replying to a subsequent request. But if the response is stale, 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.

14.9 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 is to be given in the response.

> Note that HTTP/1.0 caches might not implement `Cache-Control` and might only implement `Pragma: no-cache` (see section 14.32).

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

```
Cache-Control   = "Cache-Control" ":" 1#cache-directive
cache-directive = cache-request-directive
       | cache-response-directive
cache-request-directive =
         "no-cache"                          ; Section 14.9.1
       | "no-store"                          ; Section 14.9.2
       | "max-age" "=" delta-seconds         ; Section 14.9.3, 14.9.4
       | "max-stale" [ "=" delta-seconds ]   ; Section 14.9.3
       | "min-fresh" "=" delta-seconds       ; Section 14.9.3
       | "no-transform"                      ; Section 14.9.5
       | "only-if-cached"                    ; Section 14.9.4
       | cache-extension                     ; Section 14.9.6
 cache-response-directive =
         "public"                               ; Section 14.9.1
       | "private" [ "=" <"> 1#field-name <"> ] ; Section 14.9.1
       | "no-cache" [ "=" <"> 1#field-name <"> ]; Section 14.9.1
       | "no-store"                             ; Section 14.9.2
       | "no-transform"                         ; Section 14.9.5
       | "must-revalidate"                      ; Section 14.9.4
       | "proxy-revalidate"                     ; Section 14.9.4
```

```
        | "max-age" "=" delta-seconds            ; Section 14.9.3
        | "s-maxage" "=" delta-seconds           ; Section 14.9.3
        | cache-extension                        ; Section 14.9.6
    cache-extension = token [ "=" ( token | quoted-string ) ]
```

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 might 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 are cacheable; 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 user agent.
- Modifications of the basic expiration mechanism; these may be imposed by either the origin server or the user agent.
- Controls over cache revalidation and reload; these may only be imposed by a user agent.
- Control over transformation of entities.
- Extensions to the caching system.

14.9.1 What is Cacheable

By default, a response is cacheable if the requirements of the request method, request header fields, and the response status indicate that it is cacheable. Section 13.4 summarizes these defaults for cacheability. The following `Cache-Control` response directives allow an origin server to override the default cacheability of a response:

`public`

Indicates that the response MAY be cached by any cache, even if it would normally be non-cacheable or cacheable only within a non-shared cache. (See also `Authorization`, section 14.8, for additional details.)

`private`

Indicates that all or part of the response message is 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. A private (non-shared) cache MAY cache the response.

Note: This usage of the word `private` only controls where the response may be cached, and cannot ensure the privacy of the message content.

`no-cache`

If the `no-cache` directive does not specify a field-name, then a cache MUST NOT use the response to satisfy a subsequent request without successful revalidation with the origin server. This allows an origin server to prevent caching even by caches that have been configured to return stale responses to client requests.

If the `no-cache` directive does specify one or more field-names, then a cache MAY use the response to satisfy a subsequent request, subject to any other restrictions on caching. However, the specified field-name(s) MUST NOT be sent in the response to a subsequent request without successful revalidation with the origin server. This allows an origin server to prevent the re-use of certain header fields in a response, while still allowing caching of the rest of the response.

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

14.9.2 What May be Stored by Caches

`no-store`

The purpose of the `no-store` directive is to prevent the inadvertent release or retention of sensitive information (for example, on backup tapes). 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. “MUST NOT store” in this context means that the cache MUST NOT intentionally store the information in non-volatile storage, and MUST make a best-effort attempt to remove the information from volatile storage as promptly as possible after forwarding it.

Even when this directive is associated with a response, users might 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 might improve privacy in some cases, we caution that it is NOT in any way a reliable or sufficient mechanism for ensuring privacy. In particular, malicious or compromised caches might not recognize or obey this directive, and communications networks might be vulnerable to eavesdropping.

14.9.3 Modifications of the Basic Expiration Mechanism

The expiration time of an entity MAY be specified by the origin server using the `Expires` header (see section 14.21). Alternatively, it MAY be specified using the `max-age` directive in a response. When the `max-age` cache-control directive is present in a cached response, the response is stale if its current age is greater than the age value given (in seconds) at the time of a new request for that resource. The `max-age` directive on a response implies that the response is cacheable (i.e., “`public`”) unless some other, more restrictive cache directive is also present.

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 might be useful if certain HTTP/1.0 caches improperly calculate ages or expiration times, perhaps due to desynchronized clocks.

Many HTTP/1.0 cache implementations will treat an `Expires` value that is less than or equal to the response `Date` value as being equivalent to the `Cache-Control` response directive "`no-cache`". If an HTTP/1.1 cache receives such a response, and the response does not include a `Cache-Control` header field, it SHOULD consider the response to be non-cacheable in order to retain compatibility with HTTP/1.0 servers.

Note: An origin server might wish to use a relatively new HTTP cache control feature, such as the “`private`” directive, on a network including older caches that do not understand that feature. The origin server will need to combine the new feature with an `Expires` field whose value is less than or equal to the `Date` value. This will prevent older caches from improperly caching the response.

`s-maxage`

If a response includes an `s-maxage` directive, then for a shared cache (but not for a private cache), the maximum age specified by this directive overrides the maximum age specified by either the max-age directive or the `Expires` header. The `s-maxage` directive also implies the semantics of the `proxy-revalidate` directive (see section 14.9.4), i.e., that the shared cache must not use the entry after it becomes stale to respond to a subsequent request without first revalidating it with the origin server. The `s-`maxage directive is always ignored by a private cache.

Note that most older caches, not compliant with this specification, do not implement any cache-control directives. An origin server wishing to use a cache-control directive that restricts, but does not prevent, caching by an HTTP/1.1-

compliant cache MAY exploit the requirement that the `max-age` directive overrides the `Expires` header, and the fact that pre-HTTP/1.1-compliant caches do not observe the `max-age` directive.

Other directives allow a user agent to modify the basic expiration mechanism. 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` directive is also included, the client is not willing to accept a stale response.

`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 response that 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. If `max-stale` is assigned a value, then the client is willing to accept a response that has exceeded its expiration time by no more than the specified number of seconds. If no value is assigned to `max-stale`, then the client is willing to accept a stale response of any age.

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 110 (Response is stale).

A cache MAY be configured to return stale responses without validation, but only if this does not conflict with any "MUST"-level requirements concerning cache validation (e.g., a "`must-revalidate`" cache-control directive).

If both the new request and the cached entry include "`max-age`" directives, then the lesser of the two values is used for determining the freshness of the cached entry for that request.

14.9.4 Cache Revalidation and Reload Controls

Sometimes a user agent might 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 might 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 cache entry has become corrupted for some reason.

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 a "`no-cache`" cache-control directive or, for compatibility with HTTP/1.0 clients, "Pragma: no-cache". Field names MUST NOT 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 a "`max-age`=0" cache-control directive, 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 "`max-age`=0" cache-control directive, 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.

`max-age`

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 might 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 might 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 can return its now validated copy to the client with a 200 (OK) response. If the server replies with a new entity and cache validator, however, the intermediate cache can 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 with a 200 (OK) response.

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

`only-if-cached`

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 entry 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.

`must-revalidate`

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 entry 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, if, based solely on the origin server's `Expires` or `max-age` value, the cached response is stale.)

The `must-revalidate` directive is necessary to support reliable operation for certain 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.

Servers SHOULD send the `must-revalidate` directive if and only if failure to revalidate a request on the entity could result in 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.

`proxy-revalidate`

The `proxy-revalidate` directive has the same meaning as the `must-revalidate` directive, except that it does not apply to non-shared user agent caches. It can be used on a response to an authenticated request to permit the user's cache to store and later return the response without needing to revalidate it (since it has already been authenticated once by that user), while still requiring proxies that service many users to revalidate each time (in order to make sure that each user has been authenticated). Note that such authenticated responses also need the `public` cache control directive in order to allow them to be cached at all.

14.9.5 No-Transform Directive

`no-transform`

Implementors of intermediate caches (proxies) have found it useful to convert the media type of certain entity bodies. A non-transparent proxy might, for example, convert between image formats in order to save cache space or to reduce the amount of traffic on a slow link.

Serious operational problems occur, however, when these transformations are applied to entity bodies intended for certain kinds of applications. For example, applications for medical imaging, scientific data analysis and those using end-to-end authentication, all depend on receiving an entity body that is bit for bit identical to the original entity-body.

Therefore, if a message includes the `no-transform` directive, an intermediate cache or proxy MUST NOT change those headers that are listed in section 13.5.2 as being subject to the `no-transform` directive. This implies that the cache or proxy MUST NOT change any aspect of the entity-body that is specified by these headers, including the value of the entity-body itself.

14.9.6 Cache Control Extensions

The `Cache-Control` header field can be extended through the use of one or more cache-extension tokens, each with an optional assigned value. Informational extensions (those which do not require a change in cache behavior) MAY be added without changing the semantics of other directives. Behavioral extensions are designed to work by acting as modifiers to the existing base of cache directives. Both the new directive and the standard directive are supplied, such that applications which do not understand the new directive will default to the behavior specified by the standard directive, and those that understand the new directive will recognize it as modifying the requirements associated with the standard directive. In this way, extensions to the cache-control directives can be made without requiring changes to the base protocol.

This extension mechanism depends on an HTTP cache obeying all of the cache-control directives defined for its native HTTP-version, obeying certain extensions, and ignoring all directives that it does not understand.

For example, consider a hypothetical new response directive called `community` which acts as a modifier to the `private` directive. We define this new directive to mean that, in addition to any non-shared cache, any cache which is shared only by members of the community named within its value may cache the response. An origin server wishing to allow the `UCI` community to use an otherwise private response in their shared cache(s) could do so by including

```
Cache-Control: private, community="UCI"
```

A cache seeing this header field will act correctly even if the cache does not understand the `community` cache-extension, since it will also see and understand the `private` directive and thus default to the safe behavior.

Unrecognized cache-directives MUST be ignored; it is assumed that any cache-directive likely to be unrecognized by an HTTP/1.1 cache will be combined with standard directives (or the response's default cacheability) such that the cache behavior will remain minimally correct even if the cache does not understand the extension(s).

14.10 Connection

The `Connection` general-header field allows the sender to specify options that are desired for that particular connection and MUST NOT be communicated by proxies over further connections.

The `Connection` header has the following grammar:

```
Connection = "Connection" ":" 1#(connection-token)
connection-token  = token
```

HTTP/1.1 proxies MUST parse the `Connection` header field before a message is forwarded and, for each connection-token in this field, remove any header field(s) from the message with the same name as the connection-token. `Connection` options are signaled by the presence of a connection-token in the `Connection` header field, not by any corresponding additional header field(s), since the additional header field may not be sent if there are no parameters associated with that connection option.

Message headers listed in the `Connection` header MUST NOT include end-to-end headers, such as `Cache-Control`.

HTTP/1.1 defines the "`close`" connection option for the sender to signal that the connection will be closed after completion of the response. For example,

```
Connection: close
```

in either the request or the response header fields indicates that the connection SHOULD NOT be considered 'persistent' (section 8.1) after the current request/response is complete.

HTTP/1.1 applications that do not support persistent connections MUST include the "`close`" connection option in every message.

A system receiving an HTTP/1.0 (or lower-version) message that includes a `Connection` header MUST, for each connection-token in this field, remove and ignore any header field(s) from the message with the same name as the connection-token. This protects against mistaken forwarding of such header fields by pre-HTTP/1.1 proxies. See section 19.6.2.

14.11 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 entity-body, 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-coding is a characteristic of the entity identified by the `Request-URI`. Typically, the entity-body is stored with this encoding and is only decoded before rendering or analogous usage. However, a non-transparent proxy MAY modify the content-coding if the new coding is known to be acceptable to the recipient, unless the "`no-transform`" cache-control directive is present in the message.

If the content-coding of an entity is not "`identity`", then the response MUST include a `Content-Encoding` entity-header (section 14.11) that lists the non-identity content-coding(s) used.

If the content-coding of an entity in a request message is not acceptable to the origin server, the server SHOULD respond with a status code of 415 (Unsupported Media Type).

If multiple encodings have been applied to an entity, 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.

14.12 Content-Language

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

```
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 user to identify and differentiate entities according to the user's own preferred language. Thus, if the body content is intended only for a Danish-literate audience, the appropriate field is

```
Content-Language: da
```

If no Content-Language is specified, the default is that the content is intended for all language audiences. This might mean that the sender does not consider it to be specific to any natural language, or that the sender does not know for which language it is intended.

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

```
Content-Language: mi, en
```

However, just because multiple languages are present within 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 would properly only include "en".

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

14.13 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 transfer-length of the message-body, unless this is prohibited by the rules in section 4.4.

Any Content-Length greater than or equal to zero is a valid value. Section 4.4 describes how to determine the length of a message-body if a Content-Length is not given.

Note that 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 sent whenever the message's length can be determined prior to being transferred, unless this is prohibited by the rules in section 4.4.

14.14 Content-Location

The Content-Location entity-header field MAY be used to supply the resource location for the entity enclosed in the message when that entity is accessible from a location separate from the requested resource's URI. A server SHOULD provide a Content-Location for the variant corresponding to the response entity; especially in the case where a resource has multiple entities associated with it, and those entities actually have separate locations by which they might be individually accessed, the server SHOULD provide a Content-Location for the particular variant which is returned.

```
Content-Location = "Content-Location" ":"
                   ( absoluteURI | relativeURI )
```

The value of Content-Location also defines the base URI for the entity.

The Content-Location value is not a replacement for the original requested URI; it is only a statement of the location of the resource corresponding to this particular entity at the time of the request. Future requests MAY specify the Content-Location URI as the request-URI if the desire is to identify the source of that particular entity.

A cache cannot assume that an entity with a Content-Location different from the URI used to retrieve it can be used to respond to later requests on that Content-Location URI. However, the Content-Location can be used to differentiate between multiple entities retrieved from a single requested resource, as described in section 13.6.

If the Content-Location is a relative URI, the relative URI is interpreted relative to the Request-URI.

The meaning of the `Content-Location` header in PUT or POST requests is undefined; servers are free to ignore it in those cases.

14.15 Content-MD5

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

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

The `Content-MD5` header field MAY be generated by an origin server or client to function as an integrity check of the entity-body. Only origin servers or clients 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-coding that has been applied, but not including any transfer-encoding applied to the message-body. If the message 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.

There are several consequences of this. The entity-body for composite types MAY 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. The `Transfer-Encoding` header field is not allowed within body-parts.

Conversion of all line breaks to CRLF MUST NOT be done before computing or checking the digest: the line break convention used in the text actually transmitted MUST be left unaltered when computing the digest.

> 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 CRLF.

14.16 Content-Range

The `Content-Range` entity-header is sent with a partial entity-body to specify where in the full entity-body the partial body should be applied. Range units are defined in section 3.12.

```
Content-Range = "Content-Range" ":" content-range-spec
content-range-spec      = byte-content-range-spec
byte-content-range-spec = bytes-unit SP
                          byte-range-resp-spec "/"
                          ( instance-length | "*" )

byte-range-resp-spec = (first-byte-pos "-" last-byte-pos)
```

```
                                 | "*"
instance-length             = 1*DIGIT
```

The header SHOULD indicate the total length of the full entity-body, unless this length is unknown or difficult to determine. The asterisk "*" character means that the `instance-length` is unknown at the time when the response was generated.

Unlike `byte-ranges-specifier` values (see section 14.35.1), a `byte-range-resp-spec` MUST 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` with a `byte-range-resp-spec` whose `last-byte-pos` value is less than its `first-byte-pos value`, or whose `instance-length` value is less than or equal to 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.

A server sending a response with status code 416 (Requested range not satisfiable) SHOULD include a `Content-Range` field with a `byte-range-resp-spec` of "*". The `instance-length` specifies the current length of the selected resource. A response with status code 206 (Partial Content) MUST NOT include a `Content-Range` field with a `byte-range-resp-spec` of "*".

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`

When an HTTP message includes the content of a single range (for example, a response to a request for a single range, or to a 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/1.1 206 Partial content
Date: Wed, 15 Nov 1995 06:25:24 GMT
Last-Modified: Wed, 15 Nov 1995 04:58:08 GMT
Content-Range: bytes 21010-47021/47022
Content-Length: 26012
Content-Type: image/gif
```

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 message. The multipart media type used for this purpose is "multipart/byteranges" as defined in appendix 19.2. See appendix 19.6.3 for a compatibility issue.

A response to a request for a single range MUST NOT be sent using the multipart/byteranges media type. A response to a request for multiple ranges, whose result is a single range, MAY be sent as a multipart/byteranges media type with one part. A client that cannot decode a multipart/byteranges message MUST 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 syntactically invalid, the server SHOULD treat the request as if the invalid `Range` header field did not exist. (Normally, this means return a 200 response containing the full entity).

If the server receives a request (other than one including an `If-Range` request-header field) with an unsatisfiable `Range` request-header field (that is, all of whose `byte-range-spec` values have a `first-byte-pos` value greater than the current length of the selected resource), it SHOULD return a response code of 416 (Requested range not satisfiable) (section 10.4.17).

> Note: clients cannot depend on servers to send a 416 (Requested range not satisfiable) response instead of a 200 (OK) response for an unsatisfiable Range request-header, since not all servers implement this request-header.

14.17 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.

14.18 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 3.3.1; it MUST be sent in RFC 1123 [8]-date format.

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

An example is

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

Origin servers MUST include a `Date` header field in all responses, except in these cases:

1. If the response status code is 100 (Continue) or 101 (Switching Protocols), the response MAY include a `Date` header field, at the server's option.
2. If the response status code conveys a server error, e.g. 500 (Internal Server Error) or 503 (Service Unavailable), and it is inconvenient or impossible to generate a valid `Date`.
3. If the server does not have a clock that can provide a reasonable approximation of the current time, its responses MUST NOT include a `Date` header field. In this case, the rules in section 14.18.1 MUST be followed.

A received message that does not have a `Date` header field MUST be assigned one by the recipient if the message will be cached by that recipient or gatewayed via a protocol which requires a `Date`. An HTTP implementation without a clock MUST NOT cache responses without revalidating them on every use. An HTTP cache, especially a shared cache, SHOULD use a mechanism, such as NTP [28], to synchronize its clock with a reliable external standard.

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 client without a clock MUST NOT send a `Date` header field in a request.

The HTTP-date sent in a `Date` header SHOULD NOT represent a date and time subsequent to the generation of the message. It SHOULD represent the best available approximation of the date and time of message generation, unless the implementation has no means of generating a reasonably accurate date and time. In theory, the date ought to 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.

14.18.1 Clockless Origin Server Operation

Some origin server implementations might not have a clock available. An origin server without a clock MUST NOT assign `Expires` or `Last-Modified` values to a response, unless these values were associated with the resource by a system or user with a reliable clock. It MAY assign an `Expires` value that is known, at or before server configuration time, to be in the past (this allows “pre-expiration” of responses without storing separate `Expires` values for each resource).

14.19 ETag

The ETag response-header field provides the current value of the entity tag for the requested variant. The headers used with entity tags are described in sections 14.24, 14.26 and 14.44. The entity tag MAY be used for comparison with other entities from the same resource (see section 13.3.3).

```
ETag = "ETag" ":" entity-tag
```

Examples:

```
ETag: "xyzzy"
ETag: W/"xyzzy"
ETag: ""
```

14.20 Expect

The `Expect` request-header field is used to indicate that particular server behaviors are required by the client.

```
Expect       =  "Expect" ":" 1#expectation
expectation  =  "100-continue" | expectation-extension
expectation-extension =  token [ "=" ( token | quoted-string )
                         *expect-params ]
expect-params =  ";" token [ "=" ( token | quoted-string ) ]
```

A server that does not understand or is unable to comply with any of the expectation values in the `Expect` field of a request MUST respond with appropriate error status. The server MUST respond with a 417 (Expectation Failed) status if any of the expectations cannot be met or, if there are other problems with the request, some other 4xx status.

This header field is defined with extensible syntax to allow for future extensions. If a server receives a request containing an `Expect` field that includes an expectation-extension that it does not support, it MUST respond with a 417 (Expectation Failed) status.

Comparison of expectation values is case-insensitive for unquoted tokens (including the 100-continue token), and is case-sensitive for quoted-string expectation-extensions.

The `Expect` mechanism is hop-by-hop: that is, an HTTP/1.1 proxy MUST return a 417 (Expectation Failed) status if it receives a request with an expectation that it cannot meet. However, the `Expect` request-header itself is end-to-end; it MUST be forwarded if the request is forwarded.

Many older HTTP/1.0 and HTTP/1.1 applications do not understand the `Expect` header.

See section 8.2.3 for the use of the 100 (continue) status.

14.21 Expires

The `Expires` entity-header field gives the date/time after which the response is considered stale. A stale cache entry may not normally be returned by a cache (either a proxy cache or a user agent cache) unless it is first validated with the origin server (or with an intermediate cache that has a fresh copy of the entity). 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 3.3.1; it MUST be in RFC 1123 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 (see section 14.9.3), 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 sends an Expires date that is equal to the Date header value. (See the rules for expiration calculations in section 13.2.4.)

To mark a response as "never expires," an origin server sends an Expires date approximately one year from the time the response is sent. HTTP/1.1 servers SHOULD NOT send Expires dates more than one year in the future.

The presence of an Expires header field with a date value of some time in the future on a response that otherwise would by default be non-cacheable indicates that the response is cacheable, unless indicated otherwise by a Cache-Control header field (section 14.9).

14.22 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.

The client SHOULD NOT send the From header field without the user's approval, as it might 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.

14.23 Host

The Host request-header field specifies the Internet host and port number of the resource being requested, as obtained from the original URI 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 naming authority 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 ] ; 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/> would properly include:

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

A client MUST include a `Host` header field in all HTTP/1.1 request messages . If the requested URI does not include an Internet host name for the service being requested, then the `Host` header field MUST be given with an empty value. An HTTP/1.1 proxy MUST ensure that any request message it forwards does contain an appropriate `Host` header field that identifies the service being requested by the proxy. All Internet-based HTTP/1.1 servers MUST respond with a 400 (Bad Request) status code to any HTTP/1.1 request message which lacks a `Host` header field.

See sections 5.2 and 19.6.1.1 for other requirements relating to `Host`.

14.24 If-Match

The `If-Match` request-header field is used with a method to make it conditional. A client that has one or more entities previously obtained from the resource can verify that one of those entities is current by including a list of their associated entity tags in the `If-Match` header field. Entity tags are defined in section 3.11. 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 version of a resource. As a special case, the value “`*`” matches any current entity of the resource.

```
If-Match = "If-Match" ":" ( "*" | 1#entity-tag )
```

If any of the entity tags match the entity tag of the entity that would have been returned in the response to a similar GET request (without the `If-Match` header) on that resource, or if “`*`” is given and any current entity exists for that resource, then the server MAY perform the requested method as if the `If-Match` header field did not exist.

A server MUST use the strong comparison function (see section 13.3.3) to compare the entity tags in `If-Match`.

If none of the entity tags match, or if “`*`” is given and no current entity exists, the server MUST NOT perform the requested method, and MUST return a 412 (Precondition Failed) response. This behavior is most useful when the client wants to prevent an updating method, such as PUT, from modifying a resource that has changed since the client last retrieved it.

If the request would, without the `If-Match` header field, result in anything other than a 2xx or 412 status, then the `If-Match` header MUST be ignored.

The meaning of “`If-Match: *`” is that the method SHOULD be performed if the representation selected by the origin server (or by a cache, possibly using the `Vary` mechanism, see section 14.44) exists, and MUST NOT be performed if the representation does not exist.

A request intended to update a resource (e.g., a PUT) MAY include an `If-Match` header field to signal that the request method MUST NOT be applied if the entity corresponding to the `If-Match` value (a single entity tag) is no longer a representation of that resource. This allows the user to indicate that they do not wish the request to be successful if the resource has been changed without their knowledge. Examples:

```
If-Match: "xyzzy"
If-Match: "xyzzy", "r2d2xxxx", "c3piozzzz"
If-Match: *
```

The result of a request having both an `If-Match` header field and either an `If-None-Match` or an `If-Modified-Since` header fields is undefined by this specification.

14.25 If-Modified-Since

The `If-Modified-Since` request-header field is used with a method to make it conditional: if the requested variant has not been modified since the time specified in this field, an entity will not be returned from the server; instead, a 304 (not modified) response will be returned without any message-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 entity 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 variant has been modified since the `If-Modified-Since` date, the response is exactly the same as for a normal `GET`.

c) If the variant has not been modified since a valid `If-Modified-Since` date, the server SHOULD 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: The `Range` request-header field modifies the meaning of `If-Modified-Since`; see section 14.35 for full details.
>
> Note: `If-Modified-Since` times are interpreted by the server, whose clock might not be synchronized with the client.
>
> Note: When handling an `If-Modified-Since` header field, some servers will use an exact date comparison function, rather than a less-than function, for deciding whether to send a 304 (Not Modified) response. To get best results when sending an `If-Modified-Since` header field for cache validation, clients are advised to use the exact date string received in a previous `Last-Modified` header field whenever possible.
>
> Note: 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 encodings 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 requested and the `If-Modified-Since` date of a subsequent request, and the possibility of clock-skew-related problems if the `If-Modified-Since` 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.

The result of a request having both an `If-Modified-Since` header field and either an `If-Match` or an `If-Unmodified-Since` header fields is undefined by this specification.

14.26 If-None-Match

The `If-None-Match` request-header field is used with a method to make it conditional. A client that has one or more entities previously obtained from the resource can verify that none of those entities is current by including a list of their associated entity tags in the `If-None-Match` header field. The purpose of this feature is to allow efficient updates of cached information with a minimum amount of transaction overhead. It is also used to prevent a method (e.g. PUT) from inadvertently modifying an existing resource when the client believes that the resource does not exist.

As a special case, the value "`*`" matches any current entity of the resource.

```
If-None-Match = "If-None-Match" ":" ( "*" | 1#entity-tag )
```

If any of the entity tags match the entity tag of the entity that would have been returned in the response to a similar GET request (without the `If-None-Match` header) on that resource, or if "`*`" is given and any current entity exists for that resource, then the server MUST NOT perform the requested method, unless required to do so because the resource's modification date fails to match that supplied in an `If-Modified-Since` header field in the request. Instead, if the request method was GET or HEAD, the server SHOULD respond with a 304 (Not Modified) response, including the cache-related header fields (particularly ETag) of one of the entities that matched. For all other request methods, the server MUST respond with a status of 412 (Precondition Failed).

See section 13.3.3 for rules on how to determine if two entities tags match. The weak comparison function can only be used with GET or HEAD requests.

If none of the entity tags match, then the server MAY perform the requested method as if the `If-None-Match` header field did not exist, but MUST also ignore any `If-Modified-Since` header field(s) in the request. That is, if no entity tags match, then the server MUST NOT return a 304 (Not Modified) response.

If the request would, without the `If-None-Match` header field, result in anything other than a 2xx or 304 status, then the `If-None-Match` header MUST be ignored. (See section 13.3.4 for a discussion of server behavior when both `If-Modified-Since` and `If-None-Match` appear in the same request.)

The meaning of "`If-None-Match: *`" is that the method MUST NOT be performed if the representation selected by the origin server (or by a cache, possibly using the `Vary` mechanism, see section 14.44) exists, and SHOULD be performed if the representation does not exist. This feature is intended to be useful in preventing races between PUT operations.

Examples:

```
If-None-Match: "xyzzy"
If-None-Match: W/"xyzzy"
If-None-Match: "xyzzy", "r2d2xxxx", "c3piozzzz"
If-None-Match: W/"xyzzy", W/"r2d2xxxx", W/"c3piozzzz"
If-None-Match: *
```

The result of a request having both an `If-None-Match` header field and either an `If-Match` or an `If-Unmodified-Since` header fields is undefined by this specification.

14.27 If-Range

If a client has a partial copy of an entity in its cache, and wishes to have an up-to-date copy of the entire entity in its cache, it could use the `Range` request-header with a conditional GET (using either or both of `If-Unmodified-Since` and `If-Match`.) 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 `If-Range` 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.'

```
If-Range = "If-Range" ":" ( entity-tag | HTTP-date )
```

If the client has no entity tag for an entity, but does have a `Last-Modified` date, it MAY use that date in an `If-Range` header. (The server can distinguish between a valid `HTTP-date` and any form of `entity-tag` by examining no more than two characters.) The `If-Range` 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 entity tag given in the `If-Range` header matches the current entity tag for the entity, then the server SHOULD provide the specified sub-range of the entity using a 206 (Partial content) response. If the entity tag does not match, then the server SHOULD return the entire entity using a 200 (OK) response.

14.28 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 variant has been modified since the specified time, the server MUST NOT perform the requested operation, and MUST return a 412 (Precondition Failed).

```
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 or 412 status, the `If-Unmodified-Since` header SHOULD be ignored.

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

The result of a request having both an `If-Unmodified-Since` header field and either an `If-None-Match` or an `If-Modified-Since` header fields is undefined by this specification.

14.29 Last-Modified

The `Last-Modified` entity-header field indicates the date and time at which the origin server believes the variant was last modified.

```
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 origin server 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.

HTTP/1.1 servers SHOULD send `Last-Modified` whenever feasible.

14.30 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 (Created) 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 URI for automatic redirection to the resource. The field value consists of a single absolute URI.

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

An example is:

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

Note: The `Content-Location` header field (section 14.14) differs from `Location` in that the `Content-Location` 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`. Also see section 13.10 for cache requirements of some methods.

14.31 Max-Forwards

The `Max-Forwards` request-header field provides a mechanism with the TRACE (section 9.8) and OPTIONS (section 9.2) methods to limit the number of proxies or gateways that 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 or OPTIONS request containing a Max-Forwards header field MUST check and update its value prior to forwarding the request. If the received value is zero (0), the recipient MUST NOT forward the request; instead, it MUST respond as the final recipient. If the received Max-Forwards value is greater than zero, then the forwarded message MUST contain an updated Max-Forwards field with a value decremented by one (1).

The Max-Forwards header field MAY 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.

14.32 Pragma

The Pragma general-header field is used to include implementation-specific directives that might 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 [ "=" ( token | quoted-string ) ]
```

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 14.9) and is defined here for backward 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 might 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 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.

> Note: because the meaning of “Pragma: no-cache” as a response header field is not actually specified, it does not provide a reliable replacement for “Cache-Control: no-cache” in a response.

14.33 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-Authenticate  = "Proxy-Authenticate" ":" 1#challenge
```

The HTTP access authentication process is described in “HTTP Authentication: Basic and Digest Access Authentication” [43]. Unlike WWW-Authenticate, the Proxy-Authenticate header field applies only to the current connection and SHOULD NOT be passed on to downstream clients. However, an intermediate proxy might need to obtain its own credentials by requesting them from the downstream client, which in some circumstances will appear as if the proxy is forwarding the Proxy-Authenticate header field.

14.34 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 “HTTP Authentication: Basic and Digest Access Authentication” [43] . Unlike `Authorization`, the `Proxy-Authorization` header field applies only to the next outbound proxy that demanded authentication using the `Proxy-Authenticate` field. When multiple proxies are used in a chain, the `Proxy-Authorization` header field is consumed by the first outbound proxy that was expecting to receive credentials. A proxy MAY relay the credentials from the client request to the next proxy if that is the mechanism by which the proxies cooperatively authenticate a given request.

14.35 Range

14.35.1 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 in the entity-body (not necessarily the same as the message-body).

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 syntactically invalid. The recipient of a byte-range-set that includes one or more syntactically invalid byte-range-spec values MUST ignore the header field that includes that byte-range-set.

If the last-byte-pos value is absent, or if the value is greater than or equal to the current length of the entity-body, last-byte-pos is taken to be equal to one less than the current length of the entity-body in bytes.

By its choice of last-byte-pos, a client can limit the number of bytes retrieved without knowing the size 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-body, of a length given by the suffix-length value. (That is, this form specifies the last N bytes of an entity-body.) If the entity is shorter than the specified suffix-length, the entire entity-body is used.

If a syntactically valid byte-range-set includes at least one byte-range-spec whose first-byte-pos is less than the current length of the entity-body, or at least one suffix-byte-range-spec with a non-zero suffix-length, then the byte-range-set is satisfiable. Otherwise, the byte-range-set is unsatisfiable. If the byte-range-set is unsatisfiable, the server SHOULD return a response with a status of 416 (Requested range not satisfiable). Otherwise, the server SHOULD return a response with a status of 206 (Partial Content) containing the satisfiable ranges of the entity-body.

Examples of byte-ranges-specifier values (assuming an entity-body 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
  ```

14.35.2 Range Retrieval Requests

HTTP retrieval requests using conditional or unconditional GET methods MAY request one or more sub-ranges of the entity, instead of the entire entity, using the `Range` request header, which applies to the entity returned as the result of the request:

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

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

If 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-None-Match`, or one or both of `If-Unmodified-Since` and `If-Match`) 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 might be more appropriate to use the `If-Range` header (see section 14.27) in addition to the `Range` header.

If a proxy that supports ranges receives a `Range` request, forwards the request to an inbound server, and receives an entire entity in reply, it SHOULD only return the requested range to its client. It SHOULD store the entire received response in its cache if that is consistent with its cache allocation policies.

14.36 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 (the "referrer", although the header field is misspelled.) The `Referer` request-header 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 the field value is a relative URI, it SHOULD be interpreted relative to the `Request-URI`. The URI MUST NOT include a fragment. See section 15.1.3 for security considerations.

14.37 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. This field MAY also be used with any 3xx (Redirection) response to indicate the minimum time the user-agent is asked wait before issuing the redirected request. 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: Fri, 31 Dec 1999 23:59:59 GMT
Retry-After: 120
```

In the latter example, the delay is 2 minutes.

14.38 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. 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 modify the `Server` response-header. Instead, it SHOULD include a `Via` field (as described in section 14.45).

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

14.39 TE

The TE request-header field indicates what extension transfer-codings it is willing to accept in the response and whether or not it is willing to accept trailer fields in a chunked transfer-coding. Its value may consist of the keyword “`trailers`” and/or a comma-separated list of extension transfer-coding names with optional accept parameters (as described in section 3.6).

```
TE           = "TE" ":" #( t-codings )
t-codings   = "trailers" | ( transfer-extension [ accept-params ] )
```

The presence of the keyword “`trailers`” indicates that the client is willing to accept trailer fields in a chunked transfer-coding, as defined in section 3.6.1. This keyword is reserved for use with transfer-coding values even though it does not itself represent a transfer-coding.

Examples of its use are:

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

The `TE` header field only applies to the immediate connection. Therefore, the keyword MUST be supplied within a `Connection` header field (section 14.10) whenever TE is present in an HTTP/1.1 message.

A server tests whether a transfer-coding is acceptable, according to a TE field, using these rules:

1. The “`chunked`” transfer-coding is always acceptable. If the keyword “`trailers`” is listed, the client indicates that it is willing to accept trailer fields in the chunked response on behalf of itself and any downstream clients. The implication is that, if given, the client is stating that either all downstream clients are willing to accept trailer fields in the forwarded response, or that it will attempt to buffer the response on behalf of downstream recipients.

Note: HTTP/1.1 does not define any means to limit the size of a chunked response such that a client can be assured of buffering the entire response.

2. If the transfer-coding being tested is one of the transfer-codings listed in the TE field, then it is acceptable unless it is accompanied by a qvalue of 0. (As defined in section 3.9, a qvalue of 0 means "not acceptable.")
3. If multiple transfer-codings are acceptable, then the acceptable transfer-coding with the highest non-zero qvalue is preferred. The "`chunked`" transfer-coding always has a qvalue of 1.

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

14.40 Trailer

The `Trailer` general field value indicates that the given set of header fields is present in the trailer of a message encoded with chunked transfer-coding.

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

An HTTP/1.1 message SHOULD include a `Trailer` header field in a message using chunked transfer-coding with a non-empty trailer. Doing so allows the recipient to know which header fields to expect in the trailer.

If no `Trailer` header field is present, the trailer SHOULD NOT include any header fields. See section 3.6.1 for restrictions on the use of trailer fields in a "`chunked`" transfer-coding.

Message header fields listed in the `Trailer` header field MUST NOT include the following header fields:

- `Transfer-Encoding`
- `Content-Length`
- `Trailer`

14.41 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-coding in that the transfer-coding is a property of the message, not of the entity.

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

Transfer-codings are defined in section 3.6. An example is:

```
Transfer-Encoding: chunked
```

If multiple encodings have been applied to an entity, the transfer-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.

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

14.42 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 14.10) 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.

14.43 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. 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
```

14.44 Vary

The `Vary` field value indicates the set of request-header fields that fully determines, while the response is fresh, whether a cache is permitted to use the response to reply to a subsequent request without revalidation. For uncacheable or stale responses, the `Vary` field value advises the user agent about the criteria that were used to select the representation. A `Vary` field value of "*" implies that a cache cannot determine from the request headers of a subsequent request whether this response is the appropriate representation. See section 13.6 for use of the `Vary` header field by caches.

```
Vary  = "Vary" ":" ( "*" | 1#field-name )
```

An HTTP/1.1 server SHOULD include a `Vary` header field with any cacheable response that is subject to server-driven negotiation. Doing so allows a cache to properly interpret future requests on that resource and informs the user agent about the presence of negotiation on that resource. A server MAY include a `Vary` header field with a non-cacheable response that is subject to server-driven negotiation, since this might provide the user agent with useful information about the dimensions over which the response varies at the time of the response.

A `Vary` field value consisting of a list of field-names signals that the representation selected for the response is based on a selection algorithm which considers ONLY the listed request-header field values in selecting the most appropriate representation. A cache MAY assume that the same selection will be made for future requests with the same values for the listed field names, for the duration of time for which the response is fresh.

The field-names given are not limited to the set of standard request-header fields defined by this specification. Field names are case-insensitive.

A `Vary` field value of "*" signals that unspecified parameters not limited to the request-headers (e.g., the network address of the client), play a role in the selection of the response representation. The "*" value MUST NOT be generated by a proxy server; it may only be generated by an origin server.

14.45 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 represents each proxy or gateway that has forwarded the message. Each recipient MUST append its 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.

14.46 Warning

The `Warning` general-header field is used to carry additional information about the status or transformation of a message which might not be reflected in the message. This information is typically used to warn about a possible lack of semantic transparency from caching operations or transformations applied to the entity body of the message.

Warning headers are sent with responses using:

```
Warning    = "Warning" ":" 1#warning-value
warning-value = warn-code SP warn-agent SP warn-text
                                      [SP warn-date]

warn-code  = 3DIGIT
warn-agent = ( host [ ":" port ] ) | pseudonym
                ; the name or pseudonym of the server adding
                ; the Warning header, for use in debugging
warn-text  = quoted-string
warn-date  = <"> HTTP-date <">
```

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-8859-1.

If a character set other than ISO-8859-1 is used, it MUST be encoded in the `warn-text` using the method described in RFC 2047 [14].

Warning headers can in general be applied to any message, however some specific `warn-codes` are specific to caches and can only be applied to response messages. New `Warning` headers SHOULD be added after any existing `Warning` headers. A cache MUST NOT delete any `Warning` header that it received with a message. However, if a cache successfully validates a cache entry, it SHOULD remove any `Warning` headers previously attached to that entry except as specified for specific `Warning` codes. 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.

When multiple `Warning` headers are attached to a response, the user agent ought to inform the user of as many of them as possible, in the order that they appear in the response. If it is not possible to inform the user of 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 character set take priority over warnings in other character sets but with identical `warn-codes` and `warn-agents`.

Systems that generate multiple `Warning` headers SHOULD order them with this user agent behavior in mind.

Requirements for the behavior of caches with respect to Warnings are stated in section 13.1.2.

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

110 Response is stale
MUST be included whenever the returned response is stale.

111 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.

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

113 Heuristic expiration
MUST be included if the cache heuristically chose a freshness lifetime greater than 24 hours and the response's age is greater than 24 hours.

199 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, besides presenting the warning to the user.

214 Transformation applied
MUST be added by an intermediate cache or proxy if it applies any transformation changing the content-coding (as specified in the `Content-Encoding` header) or media-type (as specified in the `Content-Type` header) of the response, or the entity-body of the response, unless this `Warning` code already appears in the response.

299 Miscellaneous persistent 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.

If an implementation sends a message with one or more `Warning` headers whose version is HTTP/1.0 or lower, then the sender MUST include in each `warning-value` a `warn-date` that matches the date in the response.

If an implementation receives a message with a `warning-value` that includes a `warn-date`, and that `warn-date` is different from the `Date` value in the response, then that `warning-value` MUST be deleted from the message before storing, forwarding, or using it. (This prevents bad consequences of naive caching of `Warning` header fields.) If all of the `warning-values` are deleted for this reason, the `Warning` header MUST be deleted as well.

14.47 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 "HTTP Authentication: Basic and Digest Access Authentication" [43]. User agents are advised to take special care in parsing the `WWW-Authenticate` field value as it might contain more than one challenge, or if more than one `WWW-Authenticate` header field is provided, the contents of a challenge itself can contain a comma-separated list of authentication parameters.

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 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 implementors be particularly careful in this area. History shows that errors in this area often create serious security and/or privacy problems and generate highly adverse publicity for the implementor's company.

15.1.1 Abuse of Server Log Information

A server is in the position to save personal data about a user's requests which might identify their reading patterns or subjects of interest. This information is clearly confidential in nature and its handling can 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.

15.1.2 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 might allow the server machine to become more vulnerable to attacks against software that is known to contain security holes. Implementors 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` header 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` header might 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.

The `User-Agent` (section 14.43) or `Server` (section 14.38) header fields can sometimes be used to determine that a specific client or server have a particular security hole which might be exploited. Unfortunately, this same information is often used for other valuable purposes for which HTTP currently has no better mechanism.

15.1.3 Encoding Sensitive Information in URI's

Because the source of a link might be private information or might 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.

Clients SHOULD NOT include a `Referer` header field in a (non-secure) HTTP request if the referring page was transferred with a secure protocol.

Authors of services which use the HTTP protocol SHOULD NOT use GET based forms for the submission of sensitive data, because this will cause this data to be encoded in the `Request-URI`. Many existing servers, proxies, and user agents will log the request URI in some place where it might be visible to third parties. Servers can use POST-based form submission instead

15.1.4 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 or not to start sending `Accept-Language` headers to a server if it detects, by looking for any `Vary` response-header fields 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 ought to 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.

15.2 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.

15.3 DNS Spoofing

Clients using HTTP rely heavily on the Domain Name Service, and are thus generally prone to security attacks based on the deliberate mis-association of IP addresses and DNS names. 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. It is proper for these lookups to 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 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 network 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.

15.4 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.

15.5 Content-Disposition Issues

RFC 1806 [35], from which the often implemented `Content-Disposition` (see section 19.5.1) header in HTTP is derived, has a number of very serious security considerations. `Content-Disposition` is not part of the HTTP standard, but since it is widely implemented, we are documenting its use and risks for implementors. See RFC 2183 [49] (which updates RFC 1806) for details.

15.6 Authentication Credentials and Idle Clients

Existing HTTP clients and user agents typically retain authentication information indefinitely. HTTP/1.1. does not provide a method for a server to direct clients to discard these cached credentials. This is a significant defect that requires further extensions to HTTP. Circumstances under which credential caching can interfere with the application's security model include but are not limited to:

- Clients which have been idle for an extended period following which the server might wish to cause the client to reprompt the user for credentials.
- Applications which include a session termination indication (such as a 'logout' or 'commit' button on a page) after which the server side of the application 'knows' that there is no further reason for the client to retain the credentials.

This is currently under separate study. There are a number of work-arounds to parts of this problem, and we encourage the use of password protection in screen savers, idle time-outs, and other methods which mitigate the security problems inherent in this problem. In particular, user agents which cache credentials are encouraged to provide a readily accessible mechanism for discarding cached credentials under user control.

15.7 Proxies and Caching

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

Proxy operators should protect the systems on which proxies run as they would protect any system that contains or transports sensitive information. In particular, log information gathered at proxies often contains highly sensitive personal information, and/or information about organizations. Log information should be carefully guarded, and appropriate guidelines for use developed and followed. (Section 15.1.1).

Caching proxies provide additional potential vulnerabilities, since the contents of the cache represent an attractive target for malicious exploitation. Because cache contents persist after an HTTP request is complete, an attack on the cache can reveal information long after a user believes that the information has been removed from the network. Therefore, cache contents should be protected as sensitive information.

Proxy implementors should consider the privacy and security implications of their design and coding decisions, and of the configuration options they provide to proxy operators (especially the default configuration).

Users of a proxy need to be aware that they are no trustworthier than the people who run the proxy; HTTP itself cannot solve this problem.

The judicious use of cryptography, when appropriate, may suffice to protect against a broad range of security and privacy attacks. Such cryptography is beyond the scope of the HTTP/1.1 specification.

15.7.1 Denial of Service Attacks on Proxies

They exist. They are hard to defend against. Research continues. Beware.

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 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	Ross Patterson
Harald Tveit Alvestrand	Albert Lunde
Keith Ball	John C. Mallery
Brian Behlendorf	Jean-Philippe Martin-Flatin
Paul Burchard	Mitra
Maurizio Codogno	David Morris
Mike Cowlishaw	Gavin Nicol
Roman Czyborra	Bill Perry
Michael A. Dolan	Jeffrey Perry
David J. Fiander	Scott Powers
Alan Freier	Owen Rees
Marc Hedlund	Luigi Rizzo
Greg Herlihy	David Robinson
Koen Holtman	Marc Salomon
Alex Hopmann	Rich Salz
Bob Jernigan	Allan M. Schiffman
Shel Kaphan	Jim Seidman
Rohit Khare	Chuck Shotton
John Klensin	Eric W. Sink
Martijn Koster	Simon E. Spero
Alexei Kosut	Richard N. Taylor
David M. Kristol	Robert S. Thau
Daniel LaLiberte	Bill (BearHeart) Weinman
Ben Laurie	Francois Yergeau
Paul J. Leach	Mary Ellen Zurko
Daniel DuBois	Josh Cohen

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, and Larry Masinter.

Most of the specification of ranges is based on work originally done by Ari Luotonen and John Franks, with additional input from Steve Zilles.

Thanks to the "cave men" of Palo Alto. You know who you are.

Jim Gettys (the current editor of this document) wishes particularly to thank Roy Fielding, the previous editor of this document, along with John Klensin, Jeff Mogul, Paul Leach, Dave Kristol, Koen Holtman, John Franks, Josh Cohen, Alex Hopmann, Scott Lawrence, and Larry Masinter for their help. And thanks go particularly to Jeff Mogul and Scott Lawrence for performing the "MUST/MAY/SHOULD" audit.

The Apache Group, Anselm Baird-Smith, author of Jigsaw, and Henrik Frystyk implemented RFC 2068 early, and we wish to thank them for the discovery of many of the problems that this document attempts to rectify.

17 References

[1] Alvestrand, H., "Tags for the Identification of Languages" RFC 1766, March 1995.

[2] Anklesaria, F., McCahill, M., Lindner, P., Johnson, D., Torrey, D., and B. Alberti. "The Internet Gopher Protocol (a distributed document search and retrieval protocol)", RFC 1436, March 1993.

[3] Berners-Lee, T., "Universal Resource Identifiers in WWW," RFC 1630, June 1994.

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

[5] Berners-Lee, T. and D. Connolly. "Hypertext Markup Language - 2.0," RFC 1866, November 1995.

[6] Berners-Lee, T., Fielding, R. and H. Frystyk. "Hypertext Transfer Protocol -- HTTP/1.0," RFC 1945, May 1996.

[7] Freed, N., and N. Borenstein. "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies." RFC 2045, November 1996.

[8] Braden, R., "Requirements for Internet Hosts -- Communication Layers," STD 3, RFC 1123, October 1989.

[9] D. H. Crocker, "Standard for The Format of ARPA Internet Text Messages," STD 11, RFC 822, August 1982.

[10] Davis, F., Kahle, B., Morris, H., Salem, J., Shen, T., Wang, R., Sui, J., and M. Grinbaum, "WAIS Interface Protocol Prototype Functional Specification." (v1.5), Thinking Machines Corporation, April 1990.

[11] Fielding, R., "Relative Uniform Resource Locators," RFC 1808, June 1995.

[12] Horton, M., and R. Adams. "Standard for Interchange of USENET Messages," RFC 1036, December 1987.

[13] Kantor, B. and P. Lapsley. "Network News Transfer Protocol," RFC 977, February 1986.

[14] Moore, K., "MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text", RFC 2047, November 1996.

[15] Nebel, E., and L. Masinter. "Form-based File Upload in HTML," RFC 1867, November 1995.

[16] Postel, J., "Simple Mail Transfer Protocol," STD 10, RFC 821, August 1982.

[17] Postel, J., "Media Type Registration Procedure," RFC 1590, November 1996.

[18] Postel, J. and J. Reynolds. "File Transfer Protocol," STD 9, RFC 959, October 1985.

[19] Reynolds, J. and J. Postel. "Assigned Numbers," STD 2, RFC 1700, October 1994.

[20] Sollins, K. and L. Masinter. “Functional Requirements for Uniform Resource Names,” RFC 1737, 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, J., and M. Rose. “The Content-MD5 Header Field,” RFC 1864, October 1995.

[24] Carpenter, B. and Y. Rekhter. “Renumbering Needs Work,” RFC 1900, February 1996.

[25] Deutsch, P., “GZIP file format specification version 4.3,.” RFC 1952, May 1996.

[26] Venkata N. Padmanabhan, and Jeffrey C. Mogul. “Improving HTTP Latency”, Computer Networks and ISDN Systems, v. 28, pp. 25-35, Dec. 1995. Slightly revised version of paper in Proc. 2nd International WWW Conference '94: Mosaic and the Web, Oct. 1994, which is available at http://www.ncsa.uiuc.edu/SDG/IT94/Proceedings/DDay/mogul/HTTPLatency.html.

[27] Joe Touch, John Heidemann, and Katia Obraczka. “Analysis of HTTP Performance”, <URL: http://www.isi.edu/touch/pubs/http-perf96/>, ISI Research Report ISI/RR-98-463, (original report dated Aug. 1996), USC/Information Sciences Institute, August 1998.

[28] Mills, D., “Network Time Protocol (Version 3) Specification, Implementation and Analysis.” RFC 1305, March 1992.

[29] Deutsch, P., “DEFLATE Compressed Data Format Specification version 1.3.” RFC 1951, May 1996.

[30] S. Spero, “Analysis of HTTP Performance Problems,” http://sunsite.unc.edu/mdma-release/http-prob.html.

[31] Deutsch, P. and J. Gailly. “ZLIB Compressed Data Format Specification version 3.3,” RFC 1950, May 1996.

[32] Franks, J., Hallam-Baker, P., Hostetler, J., Leach, P., Luotonen, A., Sink, E., and L. Stewart. “An Extension to HTTP: Digest Access Authentication,” RFC 2069, January 1997.

[33] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., and T. Berners-Lee, “Hypertext Transfer Protocol -- HTTP/1.1”, RFC 2068, January 1997.

[34] Bradner, S., “Key words for use in RFCs to Indicate Requirement Levels,” RFC 2119, March 1997.

[35] Troost, R., and Dorner, S., “Communicating Presentation Information in Internet Messages: The Content-Disposition Header,” RFC 1806, June 1995.

[36] Mogul, J.C., Fielding, R., Gettys, J., and H. Frystyk,., “Use and Interpretation of HTTP Version Numbers”, RFC 2145, May 1997.

[37] Palme, J, “Common Internet Message Headers,” RFC 2076, February 1997.

[38] Yergeau, F., “UTF-8, a transformation format of Unicode and ISO-10646,” RFC 2279, January 1998.

[39] Nielsen, H.F., Gettys, J., Baird-Smith, A., Prud’hommeaux, E., Lie, H., and C. Lilley. “Network Performance Effects of HTTP/1.1, CSS1, and PNG,” Proceedings of ACM SIGCOMM '97, Cannes France, September 1997.

[40] Freed, N., and N. Borenstein. “Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types.” RFC 2046, November 1996.

[41] Alvestrand, H. T., “IETF Policy on Character Sets and Languages,” RFC 2277, BCP 18, January 1998.

[42] Berners-Lee, T., Fielding, R., and L. Masinter,“Uniform Resource Identifiers (URI): Generic Syntax and Semantics,” RFC 2396, August 1998.

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

[44] Luotonen, A., “Tunneling TCP based protocols through Web proxy servers,” Work in Progress.

[45] Palme, J., and A. Hopmann, “MIME E-mail Encapsulation of Aggregate Documents, such as HTML (MHTML),” RFC 2110, March 1997

[46] Bradner, S., “The Internet Standards Process -- Revision 3,” BCP 9, RFC 2026, Harvard University, October 1996.

[47] Masinter, L., “Hyper Text Coffee Pot Control Protocol (HTCPCP/1.0),” RFC 2324, April 1998.

[48] Freed, N., and N. Borenstein, “Multipurpose Internet Mail Extensions (MIME) Part Five: Conformance Criteria and Examples,” RFC 2049, November 1996.

[49] Troost, R., Dorner, S., and K. Moore, “Communicating Presentation Information in Internet Messages: The Content-Disposition Header Field,” RFC 2183, August 1997.

18 Authors’ Addresses

Roy T. Fielding
Information and Computer Science
University of California
Irvine, CA 92697-3425, USA

Fax: +1 (949) 824-1715
EMail: fielding@ics.uci.edu

James Gettys
World Wide Web Consortium
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139, USA

EMail: jg@w3.org

Jeffrey C. Mogul
Western Research Laboratory
Compaq Computer Corporation
250 University Avenue
Palo Alto, California, 94305, USA

EMail: mogul@wrl.dec.com

Henrik Frystyk Nielsen
World Wide Web Consortium
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139, USA

EMail: frystyk@w3.org

Larry Masinter
Xerox Corporation
3333 Coyote Hill Road
Palo Alto, CA 94034, USA

EMail: masinter@parc.xerox.com

Paul J. Leach
Microsoft Corporation
1 Microsoft Way
Redmond, WA 98052, USA

EMail: paulle@microsoft.com

Tim Berners-Lee
Director, World Wide Web Consortium
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139, USA

Fax: +1 (617) 258 8682
EMail: timbl@w3.org

19 Appendices

19.1 Internet Media Type message/http and application/http

In addition to defining the HTTP/1.1 protocol, this document serves as the specification for the Internet media type “message/http” and “application/http”. The message/http type can be used to enclose a single HTTP request or response message, provided that it obeys the MIME restrictions for all “message” types regarding line length and encodings. The application/http type can be used to enclose a pipeline of one or more HTTP request or response messages (not intermixed). 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
```

```
Media Type name:         application
Media subtype name:      http
Required parameters:     none
Optional parameters:     version, msgtype
 version: The HTTP-Version number of the enclosed messages
          (e.g., "1.1"). If not present, the version can be
          determined from the first line of the body.
 msgtype: The message type -- "request" or "response". If not
          present, the type can be determined from the first
          line of the body.
Encoding considerations: HTTP messages enclosed by this type
          are in "binary" format; use of an appropriate
          Content-Transfer-Encoding is required when
          transmitted via E-mail.
Security considerations: none
```

19.2 Internet Media Type multipart/byteranges

When an HTTP 206 (Partial Content) response message includes the content of multiple ranges (a response to a request for multiple non-overlapping ranges), these are transmitted as a multipart message-body. The media type for this purpose is called “multipart/byteranges”.

The multipart/byteranges media type includes two or more parts, each with its own `Content-Type` and `Content-Range` fields. The required boundary parameter specifies the boundary string used to separate each body-part.

```
Media Type name:         multipart
Media subtype name:      byteranges
Required parameters:     boundary
Optional parameters:     none
Encoding considerations: only "7bit", "8bit", or "binary" are
                         permitted
Security considerations: none
```

For example:

```
HTTP/1.1 206 Partial Content
Date: Wed, 15 Nov 1995 06:25:24 GMT
Last-Modified: Wed, 15 Nov 1995 04:58:08 GMT
Content-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--
```

Notes:

1) Additional CRLFs may precede the first boundary string in the entity.

2) Although RFC 2046 [40] permits the boundary string to be quoted, some existing implementations handle a quoted boundary string incorrectly.

3) A number of browsers and servers were coded to an early draft of the byteranges specification to use a media type of multipart/x-byteranges, which is almost, but not quite compatible with the version documented in HTTP/1.1.

19.3 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 message-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`.

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 not labeling the entity is preferred over labeling the entity with the labels US-ASCII or ISO-8859-1. See section 3.7.1 and 3.4.1.

Additional rules for requirements on parsing and encoding of dates and other potential problems with date encodings 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.
- All expiration-related calculations MUST be done in 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.

19.4 Differences Between HTTP Entities and RFC 2045 Entities

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 2045 discusses mail, and HTTP has a few features that are different from those described in RFC 2045. These differences were carefully chosen to optimize performance over binary connections, to allow greater freedom in the use of new media types, to make date comparisons easier, and to acknowledge the practice of some early HTTP servers and clients.

This appendix describes specific areas where HTTP differs from RFC 2045. 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 might be required.

19.4.1 MIME-Version

HTTP is not a MIME-compliant protocol. 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 RFC 2045[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.

19.4.2 Conversion to Canonical Form

RFC 2045 [7] requires that an Internet mail entity be converted to canonical form prior to being transferred, as described in section 4 of RFC 2049 [48]. Section 3.7.1 of this document describes the forms allowed for subtypes of the "text" media type when transmitted over HTTP. RFC 2046 requires that content with a type of "text" represent line breaks as CRLF and forbids the use of CR or LF outside of line break sequences. HTTP allows CRLF, bare CR, and bare LF to indicate a line break within text content when a message is transmitted over HTTP.

Where it is possible, a proxy or gateway from HTTP to a strict MIME environment SHOULD translate all line breaks within the text media types described in section 3.7.1 of this document to the RFC 2049 canonical form of `CRLF`. Note, however, that this might 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.

Implementors should note that conversion will break any cryptographic checksums applied to the original content unless the original content is already in canonical form. Therefore, the canonical form is recommended for any content that uses such checksums in HTTP.

19.4.3 Conversion of Date Formats

HTTP/1.1 uses a restricted set of date formats (section 3.3.1) 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.

19.4.4 Introduction of Content-Encoding

RFC 2045 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 a function equivalent to `Content-Encoding`. However, this parameter is not part of RFC 2045.)

19.4.5 No Content-Transfer-Encoding

HTTP does not use the Content-Transfer-Encoding (CTE) field of RFC 2045. 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.

19.4.6 Introduction of Transfer-Encoding

HTTP/1.1 introduces the `Transfer-Encoding` header field (section 14.41). Proxies/gateways MUST remove any transfer-coding prior to forwarding a message via a MIME-compliant protocol.

A process for decoding the "chunked" transfer-coding (section 3.6) can be represented in pseudo-code as:

```
length := 0
read chunk-size, chunk-extension (if any) 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
```

19.4.7 MHTML and Line Length Limitations

HTTP implementations which share code with MHTML [45] implementations need to be aware of MIME line length limitations. Since HTTP does not have this limitation, HTTP does not fold long lines. MHTML messages being transported by HTTP follow all conventions of MHTML, including line length limitations and folding, canonicalization, etc., since HTTP transports all message-bodies as payload (see section 3.7.2) and does not interpret the content or any MIME header lines that might be contained therein.

19.5 Additional Features

RFC 1945 and RFC 2068 document protocol elements used by some existing HTTP implementations, but not consistently and correctly across most HTTP/1.1 applications. Implementors are advised to be aware of these features, but cannot rely upon their presence in, or interoperability with, other HTTP/1.1 applications. Some of these describe proposed experimental features, and some describe features that experimental deployment found lacking that are now addressed in the base HTTP/1.1 specification.

A number of other headers, such as `Content-Disposition` and `Title`, from SMTP and MIME are also often implemented (see RFC 2076 [37]).

19.5.1 Content-Disposition

The `Content-Disposition` response-header field has been proposed as a means for the origin server to suggest a default filename if the user requests that the content is saved to a file. This usage is derived from the definition of `Content-Disposition` in RFC 1806 [35].

```
content-disposition = "Content-Disposition" ":"
                      disposition-type *( ";" disposition-parm )
disposition-type = "attachment" | disp-extension-token
disposition-parm = filename-parm | disp-extension-parm
filename-parm = "filename" "=" quoted-string
disp-extension-token = token
disp-extension-parm = token "=" ( token | quoted-string )
```

An example is

```
Content-Disposition: attachment; filename="fname.ext"
```

The receiving user agent SHOULD NOT respect any directory path information present in the `filename-parm` parameter, which is the only parameter believed to apply to HTTP implementations at this time. The filename SHOULD be treated as a terminal component only.

If this header is used in a response with the `application/octet-stream` content-type, the implied suggestion is that the user agent should not display the response, but directly enter a ‘save response as...’ dialog.

See section 15.5 for `Content-Disposition` security issues.

19.6 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. It is worth noting that, at the time of composing this specification (1996), 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. Some implementations implement the `Keep-Alive` version of persistent connections described in section 19.7.1 of RFC 2068 [33].

19.6.1 Changes from HTTP/1.0

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

19.6.1.1 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 (section 14.23) is missing from an HTTP/1.1 request, and accept absolute URIs (section 5.1.2) are among the most important changes defined by this specification.

Older HTTP/1.0 clients assumed a one-to-one relationship of IP addresses and servers; there was no other established mechanism for distinguishing the intended server of a request than the IP address to which that request was directed. The changes outlined above will allow the Internet, once older HTTP clients 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 allocated for the sole purpose of allowing special-purpose domain names to be used in root-level HTTP URLs. Given the rate of growth of the Web, and the number of servers already deployed, it is extremely important that all implementations of HTTP (including updates to existing HTTP/1.0 applications) correctly implement these requirements:

- Both clients and servers MUST support the `Host` request-header.
- A client that sends an HTTP/1.1 request MUST send a `Host` header.
- Servers MUST report a 400 (Bad Request) error if an HTTP/1.1 request does not include a `Host` request-header.
- Servers MUST accept absolute URIs.

19.6.2 Compatibility with HTTP/1.0 Persistent Connections

Some clients and servers might wish to be compatible with some previous implementations of persistent connections in HTTP/1.0 clients and servers. Persistent connections in HTTP/1.0 are explicitly negotiated as they are not the default behavior. HTTP/1.0 experimental implementations of persistent connections are faulty, and the new facilities in HTTP/1.1 are designed to rectify these problems. The problem was that some existing 1.0 clients may be sending `Keep-Alive` to a proxy server that doesn't understand `Connection`, which would then erroneously forward it to the next inbound server, which would establish the `Keep-Alive` connection and result in a hung HTTP/1.0 proxy waiting for the close on the response. The result is that HTTP/1.0 clients must be prevented from using `Keep-Alive` when talking to proxies.

However, talking to proxies is the most important use of persistent connections, so that prohibition is clearly unacceptable. Therefore, we need some other mechanism for indicating a persistent connection is desired, which is safe to use even when talking to an old proxy that ignores `Connection`. Persistent connections are the default for HTTP/1.1 messages; we introduce a new keyword (`Connection: close`) for declaring non-persistence. See section 14.10.

The original HTTP/1.0 form of persistent connections (the `Connection: Keep-Alive` and `Keep-Alive` header) is documented in RFC 2068. [33]

19.6.3 Changes from RFC 2068

This specification has been carefully audited to correct and disambiguate key word usage; RFC 2068 had many problems in respect to the conventions laid out in RFC 2119 [34].

Clarified which error code should be used for inbound server failures (e.g. DNS failures). (Section 10.5.5)

CREATE had a race that required an `Etag` be sent when a resource is first created. (Section 10.2.2)

`Content-Base` was deleted from the specification: it was not implemented widely, and there is no simple, safe way to introduce it without a robust extension mechanism. In addition, it is used in a similar, but not identical fashion in MHTML [45].

Transfer-coding and message lengths all interact in ways that required fixing exactly when chunked encoding is used (to allow for transfer encoding that may not be self delimiting); it was important to straighten out exactly how message lengths are computed. (Sections 3.6, 4.4, 7.2.2, 13.5.2, 14.13, 14.16)

A content-coding of "`identity`" was introduced, to solve problems discovered in caching. (Section 3.5)

Quality Values of zero should indicate that "I don't want something" to allow clients to refuse a representation. (Section 3.9)

The use and interpretation of HTTP version numbers has been clarified by RFC 2145. Require proxies to upgrade requests to highest protocol version they support to deal with problems discovered in HTTP/1.0 implementations (Section 3.1).

Charset wildcarding is introduced to avoid explosion of character set names in accept headers. (Section 14.2)

A case was missed in the `Cache-Control` model of HTTP/1.1; `s-maxage` was introduced to add this missing case. (Sections 13.4, 14.8, 14.9, 14.9.3)

The `Cache-Control: max-age` directive was not properly defined for responses. (Section 14.9.3)

There are situations where a server (especially a proxy) does not know the full length of a response but is capable of serving a byterange request. We therefore need a mechanism to allow byteranges with a content-range not indicating the full length of the message. (Section 14.16)

Range request responses would become very verbose if all meta-data were always returned; by allowing the server to only send needed headers in a 206 response, this problem can be avoided. (Section 10.2.7, 13.5.3, and 14.27)

Fix problem with unsatisfiable range requests; there are two cases: syntactic problems, and range doesn't exist in the document. The 416 status code was needed to resolve this ambiguity needed to indicate an error for a byte range request that falls outside of the actual contents of a document. (Section 10.4.17, 14.16)

Rewrite of message transmission requirements to make it much harder for implementors to get it wrong, as the consequences of errors here can have significant impact on the Internet, and to deal with the following problems:

1. Changing "HTTP/1.1 or later" to "HTTP/1.1", in contexts where this was incorrectly placing a requirement on the behavior of an implementation of a future version of HTTP/1.x

2. Made it clear that user-agents should retry requests, not “clients” in general.
3. Converted requirements for clients to ignore unexpected 100 (Continue) responses, and for proxies to forward 100 responses, into a general requirement for 1xx responses.
4. Modified some TCP-specific language, to make it clearer that non-TCP transports are possible for HTTP.
5. Require that the origin server MUST NOT wait for the request body before it sends a required 100 (Continue) response.
6. Allow, rather than require, a server to omit 100 (Continue) if it has already seen some of the request body.
7. Allow servers to defend against denial-of-service attacks and broken clients.

This change adds the `Expect` header and 417 status code. The message transmission requirements fixes are in sections 8.2, 10.4.18, 8.1.2.2, 13.11, and 14.20.

Proxies should be able to add `Content-Length` when appropriate. (Section 13.5.2)

Clean up confusion between 403 and 404 responses. (Section 10.4.4, 10.4.5, and 10.4.11)

Warnings could be cached incorrectly, or not updated appropriately. (Section 13.1.2, 13.2.4, 13.5.2, 13.5.3, 14.9.3, and 14.46). Warning also needed to be a general header, as PUT or other methods may have need for it in requests.

Transfer-coding had significant problems, particularly with interactions with chunked encoding. The solution is that transfer-codings become as full fledged as content-codings. This involves adding an IANA registry for transfer-codings (separate from content codings), a new header field (`TE`) and enabling trailer headers in the future. Transfer encoding is a major performance benefit, so it was worth fixing [39]. `TE` also solves another, obscure, downward interoperability problem that could have occurred due to interactions between authentication trailers, chunked encoding and HTTP/1.0 clients.(Section 3.6, 3.6.1, and 14.39)

The PATCH, LINK, UNLINK methods were defined but not commonly implemented in previous versions of this specification. See RFC 2068 [33].

The `Alternates`, `Content-Version`, `Derived-From`, `Link`, `URI`, `Public` and `Content-Base` header fields were defined in previous versions of this specification, but not commonly implemented. See RFC 2068 [33].

20 Full Copyright Statement

20.1 Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.

21 Index

While some care was taken producing this index, there is no guarantee that all occurrences of an index term have been entered into the index. Bold face italic is used for the definition of a term.

ERROR: undefined
OFFENDING COMMAND: .

STACK: