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

Copyright (C) The Internet Society (1999). All Rights Reserved.

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

HYPERTEXT TRANSFER PROTOCOL -- HTTP/1.1.........................................................1

Status of this Memo ........................................................................................................1

Copyright Notice.............................................................................................................1

Abstract .......................................................................................................................1

Table of Contents .........................................................................................................2

## 1 Introduction .........................................................................................................7

1.1 Purpose ...............................................................................................................7
1.2 Requirements ........................................................................................................7
1.3 Terminology ..........................................................................................................8
1.4 Overall Operation .................................................................................................10

## 2 Notational Conventions and Generic Grammar ..................................................11

2.1 Augmented BNF .................................................................................................11
2.2 Basic Rules ..........................................................................................................12

## 3 Protocol Parameters .........................................................................................13

3.1 HTTP Version ....................................................................................................13
3.2 Uniform Resource Identifiers ..........................................................................14
3.2.1 General Syntax ...............................................................................................14
3.2.2 http URL ........................................................................................................14
3.2.3 URI Comparison ..............................................................................................15
3.3 Date/Time Formats .............................................................................................15
3.3.1 Full Date ........................................................................................................15
3.3.2 Delta Seconds .................................................................................................16
3.4 Character Sets ....................................................................................................16
3.4.1 Missing Charset ..............................................................................................16
3.5 Content Codings .................................................................................................16
3.6 Transfer Codings ...............................................................................................17
3.6.1 Chunked Transfer Coding .............................................................................18
3.7 Media Types .......................................................................................................18
3.7.1 Canonicalization and Text Defaults ...............................................................19
3.7.2 Multipart Types ............................................................................................19
3.8 Product Tokens ..................................................................................................20
3.9 Quality Values ...................................................................................................20
3.10 Language Tags ..................................................................................................20
3.11 Entity Tags .........................................................................................................20
3.12 Range Units ......................................................................................................21

## 4 HTTP Message ..................................................................................................21

4.1 Message Types ...................................................................................................21
4.2 Message Headers ...............................................................................................21
4.3 Message Body ....................................................................................................22
4.4 Message Length ...............................................................................................23
4.5 General Header Fields .....................................................................................23
## Request

5.1 Request-Line

5.1.1 Method

5.1.2 Request-URI

5.2 The Resource Identified by a Request

5.3 Request Header Fields

## Response

6.1 Status-Line

6.1.1 Status Code and Reason Phrase

6.2 Response Header Fields

## Entity

7.1 Entity Header Fields

7.2 Entity Body

7.2.1 Type

7.2.2 Entity Length

## Connections

8.1 Persistent Connections

8.1.1 Purpose

8.1.2 Overall Operation

8.1.3 Proxy Servers

8.1.4 Practical Considerations

8.2 Message Transmission Requirements

8.2.1 Persistent Connections and Flow Control

8.2.2 Monitoring Connections for Error Status Messages

8.2.3 Use of the 100 (Continue) Status

8.2.4 Client Behavior if Server Prematurely Closes Connection

## Method Definitions

9.1 Safe and Idempotent Methods

9.1.1 Safe Methods

9.1.2 Idempotent Methods

9.2 OPTIONS

9.3 GET

9.4 HEAD

9.5 POST

9.6 PUT

9.7 DELETE

9.8 TRACE

9.9 CONNECT

## Status Code Definitions

10.1 Informational 1xx

10.1.1 100 Continue

10.1.2 101 Switching Protocols

10.2 Successful 2xx

10.2.1 200 OK

10.2.2 201 Created

10.2.3 202 Accepted

10.2.4 203 Non-Authoritative Information

10.2.5 204 No Content

10.2.6 205 Reset Content

10.2.7 206 Partial Content
10.3  Redirection 3xx.............................................................................................................. 40
  10.3.1  300 Multiple Choices................................................................................................. 40
  10.3.2  301 Moved Permanently............................................................................................ 40
  10.3.3  302 Found.................................................................................................................... 40
  10.3.4  303 See Other ............................................................................................................ 41
  10.3.5  304 Not Modified ....................................................................................................... 41
  10.3.6  305 Use Proxy........................................................................................................... 41
  10.3.7  306 (Unused) ............................................................................................................. 41
  10.3.8  307 Temporary Redirect......................................................................................... 42

10.4  Client Error 4xx ............................................................................................................ 42
  10.4.1  400 Bad Request......................................................................................................... 42
  10.4.2  401 Unauthorized....................................................................................................... 42
  10.4.3  402 Payment Required............................................................................................... 42
  10.4.4  403 Forbidden............................................................................................................ 42
  10.4.5  404 Not Found............................................................................................................ 43
  10.4.6  405 Method Not Allowed ......................................................................................... 43
  10.4.7  406 Not Acceptable .................................................................................................. 43
  10.4.8  407 Proxy Authentication Required......................................................................... 43
  10.4.9  408 Request Timeout ............................................................................................... 43
  10.4.10 409 Conflict............................................................................................................. 43
  10.4.11 410 Gone ................................................................................................................ 44
  10.4.12 411 Length Required .............................................................................................. 44
  10.4.13 412 Precondition Failed ......................................................................................... 44
  10.4.14 413 Request Entity Too Large ................................................................................. 44
  10.4.15 414 Request-URI Too Long ..................................................................................... 44
  10.4.16 415 Unsupported Media Type .................................................................................. 44
  10.4.17 416 Requested Range Not Satisfiable .................................................................... 44
  10.4.18 417 Expectation Failed ........................................................................................... 45

10.5  Server Error 5xx ............................................................................................................ 45
  10.5.1  500 Internal Server Error............................................................................................ 45
  10.5.2  501 Not Implemented ............................................................................................... 45
  10.5.3  502 Bad Gateway...................................................................................................... 45
  10.5.4  503 Service Unavailable ......................................................................................... 45
  10.5.5  504 Gateway Timeout.............................................................................................. 45
  10.5.6  505 HTTP Version Not Supported ........................................................................... 45

11  Access Authentication........................................................................................................ 46

12  Content Negotiation......................................................................................................... 46
  12.1  Server-driven Negotiation......................................................................................... 46
  12.2  Agent-driven Negotiation ......................................................................................... 47
  12.3  Transparent Negotiation ........................................................................................... 47

13  Caching in HTTP............................................................................................................ 47
  13.1.1  Cache Correctness.................................................................................................... 48
  13.1.2  Warnings................................................................................................................ 49
  13.1.3  Cache-control Mechanisms..................................................................................... 49
  13.1.4  Explicit User Agent Warnings ................................................................................ 49
  13.1.5  Exceptions to the Rules and Warnings .................................................................. 50
  13.1.6  Client-controlled Behavior .................................................................................... 50
  13.2  Expiration Model........................................................................................................ 50
    13.2.1  Server-Specified Expiration .................................................................................... 50
    13.2.2  Heuristic Expiration .............................................................................................. 51
    13.2.3  Age Calculations ................................................................................................... 51
    13.2.4  Expiration Calculations ........................................................................................ 52
14 Header Field Definitions ........................................................................................................ 62

14.1 Accept ................................................................................................................................. 62
14.2 Accept-Charset .................................................................................................................... 64
14.3 Accept-Encoding ................................................................................................................... 64
14.4 Accept-Language .................................................................................................................. 65
14.5 Accept-Ranges ..................................................................................................................... 66
14.6 Age .................................................................................................................................. 66
14.7 Allow ................................................................................................................................ 66
14.8 Authorization ....................................................................................................................... 66
14.9 Cache-Control ..................................................................................................................... 67
14.9.1 What is Cacheable ........................................................................................................... 68
14.9.2 What May be Stored by Caches ...................................................................................... 69
14.9.3 Modifications of the Basic Expiration Mechanism ........................................................ 69
14.9.4 Cache Revalidation and Reload Controls ...................................................................... 70
14.9.5 No-Transform Directive ................................................................................................. 72
14.9.6 Cache Control Extensions ............................................................................................... 72
14.10 Connection .......................................................................................................................... 72
14.11 Content-Encoding .............................................................................................................. 73
14.12 Content-Language .............................................................................................................. 73
14.13 Content-Length ................................................................................................................... 74
14.14 Content-Location ................................................................................................................ 74
14.15 Content-MD5 ..................................................................................................................... 75
14.16 Content-Range .................................................................................................................... 75
14.17 Content-Type ...................................................................................................................... 77
14.18 Date .................................................................................................................................. 77
14.18.1 Clockless Origin Server Operation .................................................................................... 78
14.19 ETag .................................................................................................................................. 78
14.20 Expect ................................................................................................................................. 78
14.21 Expires ................................................................................................................................. 78
14.22 From .................................................................................................................................. 79
14.23 Host .................................................................................................................................. 79
14.24 If-Match ............................................................................................................................... 80

Fielding, et al Standards Track [Page 5]
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
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.

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].

\[
\text{TEXT} = \langle\text{any OCTET except CTLs, but including LWS}\rangle
\]

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.

\[
\text{HEX} = \text{"A" | "B" | "C" | "D" | "E" | "F" | "a" | "b" | "c" | "d" | "e" | "f" | DIGIT}
\]

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).

\[
\text{token} = 1*\langle\text{any CHAR except CTLS or separators}\rangle
\]

\[
\text{separators} = \text{"(" | ")" | "," | ";" | ":" | "{" | "}" | ":" | "+" | "/" | ":[" | "]" | ":=" | ":=" | \}
\]

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.

\[
\text{comment} = \text{"(" * (ctext | quoted-pair | comment ) ")"}
\]

\[
\text{ctext} = \langle\text{any TEXT excluding "," and "}\rangle
\]

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

\[
\text{quoted-string} = (\text{"} * (qdtext | quoted-pair ) \text{"})
\]

\[
\text{qdtext} = \langle\text{any TEXT except \"}\rangle
\]

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

\[
\text{quoted-pair} = \text{\"\" } \text{CHAR}
\]

### 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.

\[
\text{HTTP-Version} = \text{"HTTP" } /\text{" " } 1*\text{DIGIT } "./" 1*\text{DIGIT}
\]

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"
```
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.

\[
delta-seconds = 1 \times \text{DIGIT}
\]

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].

\[
\text{charset} = \text{token}
\]

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.

\[
\text{content-coding} = \text{token}
\]

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
```

Fielding, et al Standards Track [Page 18]
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”.

Fielding, et al Standards Track [Page 19]
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.

\[
\text{product} \quad = \quad \text{token} \ ["/" \ \text{product-version}]
\]
\[
\text{product-version} \quad = \quad \text{token}
\]

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.

\[
\text{qvalue} \quad = \quad ( \ "0" \ [ \ "." \ 0*3\text{DIGIT} \ ] )
\]
\[
\mid ( \ "1" \ [ \ "." \ 0*3("0") \ ] )
\]

“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:

\[
\text{language-tag} \quad = \quad \text{primary-tag} \ (*\ (-" \ \text{subtag})\ )
\]
\[
\text{primary-tag} \quad = \quad 1*8\text{ALPHA}
\]
\[
\text{subtag} \quad = \quad 1*8\text{ALPHA}
\]

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
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

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 = 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.

\[
\begin{aligned}
\text{response-header} &= \text{Accept-Ranges} ; \text{Section 14.5} \\
&\quad | \text{Age} ; \text{Section 14.6} \\
&\quad | \text{ETag} ; \text{Section 14.19} \\
&\quad | \text{Location} ; \text{Section 14.30} \\
&\quad | \text{Proxy-Authenticate} ; \text{Section 14.33} \\
&\quad | \text{Retry-After} ; \text{Section 14.37} \\
&\quad | \text{Server} ; \text{Section 14.38} \\
&\quad | \text{Vary} ; \text{Section 14.44} \\
&\quad | \text{WWW-Authenticate} ; \text{Section 14.47}
\end{aligned}
\]

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.

\[
\begin{aligned}
\text{entity-header} &= \text{Allow} \quad ; \text{Section 14.7} \\
&\quad | \text{Content-Encoding} \quad ; \text{Section 14.11} \\
&\quad | \text{Content-Language} \quad ; \text{Section 14.12} \\
&\quad | \text{Content-Length} \quad ; \text{Section 14.13} \\
&\quad | \text{Content-Location} \quad ; \text{Section 14.14} \\
&\quad | \text{Content-MD5} \quad ; \text{Section 14.15} \\
&\quad | \text{Content-Range} \quad ; \text{Section 14.16} \\
&\quad | \text{Content-Type} \quad ; \text{Section 14.17} \\
&\quad | \text{Expires} \quad ; \text{Section 14.21} \\
&\quad | \text{Last-Modified} \quad ; \text{Section 14.29}
\end{aligned}
\]

\[\text{extension-header} = \text{message-header}\]
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.

\[
\text{entity-body} = *\text{OCTET}
\]

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:

\[
\text{entity-body} := \text{Content-Encoding} ( \text{Content-Type} ( \text{data} ) )
\]

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.