HTTP Working Group
INTERNET-DRAFT
<draft-ietf-http-v11-spec-00.ps>
Expires May 22, 1996

R. Fielding, UC Irvine
H. Frystyk, MIT/LCS
T. Berners-Lee, MIT/LCS
November 22, 1995

Hypertext Transfer Protocol — HTTP/1.1

Status of this Memo

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

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress".

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

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

Abstract

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

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

Table of Contents

1. Introduction

1.1 Purpose

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

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

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

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

1.2 Requirements

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

must
: This word or the adjective "required" means that the item is an absolute requirement of the specification.

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

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

An implementation is not compliant if it fails to satisfy one or more of the *must* requirements for the protocols it implements. An implementation that satisfies all the *must* and all the *should* requirements for its protocols is said to be "unconditionally compliant"; one that satisfies all the *must* requirements but not all the *should* requirements for its protocols is said to be "conditionally compliant".

1.3 Terminology

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

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

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

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

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

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

entity
: A particular representation or rendition of a data resource, or reply from a service resource, that may be enclosed within a request or response message. An entity consists of metainformation in the form of entity headers and content in the form of an entity body.

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

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

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

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

proxy

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

gateway

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

tunnel

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

cache

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

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

1.4 Overall Operation

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

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

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

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

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

The figure above shows three intermediaries (A, B, and C) between the user agent and origin server. A request or response message that travels the whole chain must pass through four separate connections. This distinction is important because some HTTP communication options may apply only to the connection with the nearest, non-tunnel neighbor, only to the end-points of the chain, or to all connections along the chain. Although the diagram is linear, each participant may be engaged in multiple, simultaneous communications. For example, B may be receiving requests from many clients other than A, and/or forwarding requests to servers other than C, at the same time that it is handling A's request.

Any party to the communication which is not acting as a tunnel may employ an internal cache for handling requests. The effect of a cache is that the request/response chain is shortened if one of the participants along the chain has a cached response applicable to that request. The following illustrates the resulting chain if B has a cached copy of an earlier response from O (via C) for a request which has not been cached by UA or A.

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

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

On the Internet, HTTP communication generally takes place over TCP/IP connections. The default port is TCP 80 [19], but other ports can be used. This does not preclude HTTP from being implemented on top of any other protocol on the Internet, or on other networks. HTTP only presumes a reliable transport; any protocol that provides such guarantees can be used, and 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.

For most implementations, each connection is established by the client prior to the request and closed by the server after sending the response. However, this is not a feature of the protocol and is not required by this specification. Both clients and servers must be capable of handling cases where either party closes the connection prematurely, due to user action, automated time-out, or program failure. In any case, the closing of the connection by either or both parties always terminates the current request, regardless of its status.

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 "=". Whitespace is only significant in that indentation of continuation lines is used to indicate a rule definition that spans more than one line. Certain basic rules are in uppercase, such as SP, LWS, HT, CRLF, DIGIT, ALPHA, etc. Angle brackets are used within definitions whenever their presence will facilitate discerning the use of rule names.

"literal"

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

rule1 | rule2

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

(rule1 rule2)

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

*rule

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

[rule]

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

N rule

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

#rule

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

; comment

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

implied *LWS

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

2.2 Basic Rules

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

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

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

```
CRLF           = CR LF
```

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

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

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

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

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

Hexadecimal numeric characters are used in several protocol elements.

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

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

```
word           = token | quoted-string

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

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

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

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

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

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

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

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

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

Braces are used to delimit an attribute-value bag, which may consist of a set, list, or recursively defined tokens and quoted strings. The bag semantics are defined by its context and the bag name, which may be a Uniform Resource Identifier (Section 3.2) in some fields.

```
bag            = "{" bagname 1*LWS *bagitem "}"
bagname        = token | URI
bagitem        = bag | token | quoted-string
```

3. Protocol Parameters

3.1 HTTP Version

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

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

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

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

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

HTTP/1.1 servers must:

- recognize the format of the Request-Line for HTTP/0.9, 1.0, and 1.1 requests;
- understand any valid request in the format of HTTP/0.9, 1.0, or 1.1;
- respond appropriately with a message in the same major version used by the client.

HTTP/1.1 clients must:

- recognize the format of the Status-Line for HTTP/1.0 and 1.1 responses;
- understand any valid response in the format of HTTP/0.9, 1.0, or 1.1.

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

3.2 Uniform Resource Identifiers

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

3.2.1 General Syntax

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

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

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

relativeURI    = net_path | abs_path | rel_path

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

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

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

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

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

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

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

3.2.2 http URL

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

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

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

port           = *DIGIT
```

If the port is empty or not given, port 80 is assumed. The semantics are that the identified resource is located at the server listening for TCP connections on that port of that host, and the Request-URI for the resource is abs_path. If the abs_path is not present in the URL, it must be given as "/" when used as a Request-URI for a resource (Section 5.1.2).

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

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

3.3 Date/Time Formats

3.3.1 Full Date

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

```
Sun, 06 Nov 1994 08:49:37 GMT    ; RFC 822, updated by RFC 1123
Sunday, 06-Nov-94 08:49:37 GMT   ; RFC 850, 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, though they must only generate the RFC 1123 format for representing date/time stamps in HTTP message fields.

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

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

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

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

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

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

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

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

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

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

3.3.2 Delta Seconds

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

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

3.4 Character Sets

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

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

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

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

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

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

3.5 Content Codings

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

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

> ***Note:*** *For historical reasons, HTTP applications should consider "x-gzip" and "x-compress" to be equivalent to "gzip" and "compress", respectively.*

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

gzip

An encoding format produced by the file compression program "gzip" (GNU zip) developed by Jean-loup Gailly. This format is typically a Lempel-Ziv coding (LZ77) with a 32 bit CRC. Gzip is available from the GNU project at <URL:ftp://prep.ai.mit.edu/pub/gnu/>.

compress

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

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

3.6 Transfer Codings

Transfer coding values are used to indicate an encoding transformation that has been, can be, or may need to be applied to an Entity-Body in order to ensure safe transport through the network. This differs from a content coding in that the transfer coding is a property of the message, not of the original resource.

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

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

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

All HTTP/1.1 applications must be able to receive and decode the "chunked" transfer coding. The chunked encoding modifies the body of a message in order to transfer it as a series of chunks, each with its own size indicator, followed by an optional footer containing entity-header fields. This allows dynamically-produced content to be transferred along with the information necessary for the recipient to verify that it has received the full message.

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

chunk          = chunk-size CRLF
                 chunk-data CRLF

chunk-size     = hex-no-zero *HEX
chunk-data     = chunk-size(OCTET)

footer         = *<Entity-Header, excluding Content-Length
                   and Transfer-Encoding>

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

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

3.7 Media Types

HTTP uses Internet Media Types [17] in the Content-Type (Section 10.15) and Accept (Section 10.1) header fields in order to provide open and extensible data typing and type negotiation. For mail applications, where there is no type negotiation between sender and recipient, it is reasonable to put strict limits on the set of allowed media types. With HTTP, where the sender and recipient can communicate directly, applications are allowed more freedom in the use of non-registered types. The following grammar for media types is a superset of that for MIME because it does not restrict itself to the official IANA and x-token types.

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

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

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

The type, subtype, and parameter attribute names are case-insensitive. Parameter values may or may not be case-sensitive, depending on the semantics of the parameter name. LWS should not be generated between the type and subtype, nor between an attribute and its value.

If a given media-type value has been registered by the IANA, any use of that value must be indicative of the registered data format. Although HTTP allows the use of non-registered media types, such usage must not conflict with the IANA registry. Data providers are strongly encouraged to register their media types with IANA via the procedures outlined in RFC 1590 [17].

All media-type's registered by IANA must be preferred over extension tokens. However, HTTP does not limit applications to the use of officially registered media types, nor does it encourage the use of an "x-" prefix for unofficial types outside of explicitly short experimental use between consenting applications.

3.7.1 Canonicalization and Text Defaults

Media types are registered in a canonical form. In general, entity bodies transferred via HTTP must be represented in the appropriate canonical form prior to transmission. If the body has been encoded via a Content-Encoding and/or Transfer-Encoding, the data must be in canonical form prior to that encoding. However, HTTP modifies the canonical form requirements for media of primary type "text" and for "application" types consisting of text-like records.

HTTP redefines the canonical form of text media to allow multiple octet sequences to indicate a text line break. In addition to the preferred form of CRLF, HTTP applications must accept a bare CR or LF alone as representing a single line break in text media. Furthermore, if the text media is represented in a character set which 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 sequence(s) is defined by that character set to represent the equivalent of CRLF, bare CR, and bare LF. It is assumed that any recipient capable of using such a character set will know the appropriate octet sequence for representing line breaks within that character set.

> ***Note:*** *This interpretation of line breaks applies only to the contents of an Entity-Body and only after any Transfer-Encoding and/or Content-Encoding has been removed. All other HTTP constructs use CRLF exclusively to indicate a line break. Content and transfer codings define their own line break requirements.*

A recipient of an HTTP text entity should translate the received entity line breaks to the local line break conventions before saving the entity external to the application and its cache; whether this translation takes place immediately upon receipt of the entity, or only when prompted by the user, is entirely up to the individual application.

HTTP also redefines the default character set for text media in an entity body. If a textual media type defines a charset parameter with a registered default value of "US-ASCII", HTTP changes the default to be "ISO-8859-1". Since the ISO-8859-1 [22] character set is a superset of US-ASCII [21], this does not affect the interpretation of entity bodies which only contain octets within the US-ASCII character set (0 - 127). The presence of a charset parameter value in a Content-Type header field overrides the default.

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

3.7.2 Multipart Types

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

In HTTP, multipart body-parts may contain header fields which are significant to the meaning of that part. A URI entity-header field (Section 10.42) should be included in the body-part for each enclosed entity that can be identified by a URI.

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. The following subtypes have been defined:

multipart/mixed
: The mixed subtype is used when there are no additional semantics implied beyond the fact that one or more entities are encaspsulated. HTTP servers should not use this type to send groups of entities if it is possible for those entities to be individually retrieved and cached.

multipart/alternative
: The alternative subtype implies that each of the parts is an alternative format for the same information; the user agent should present only the part most preferred by the user. HTTP servers should use some form of content negotiation (Section 12) instead of this type.

multipart/digest
: The digest subtype implies that each of the parts is a message (normally of type "message/rfc822") and thus the whole entity is a collected sequence of message traffic. This type does not have any special significance for HTTP.

multipart/form-data

The form-data subtype is defined by RFC 1867 [15] for use in submitting the data that comes about from filling-in a form.

multipart/parallel

The parallel subtype implies that the parts should be presented simultaneously by the user agent. This media type would be appropriate for situations where simultaneous presentation is an important aspect of the information, such as for audio-annotated slides.

***Note:** This document does not define what is meant by "simultaneous presentation". That is, HTTP does not provide any means of synchronization between the parts in messages of type "multipart/parallel".*

Other multipart subtypes may be registered by IANA [19] according to the procedures defined in RFC 1590 [17]. If an application receives an unrecognized multipart subtype, the application must treat it as being equivalent to "multipart/mixed".

3.8 Product Tokens

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

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

Examples:

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

Server: Apache/0.8.4
```

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

3.9 Quality Values

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

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

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

3.10 Language Tags

A language tag identifies a natural language spoken, written, or otherwise conveyed by human beings for communication of information to other human beings. Computer languages are explicitly excluded. HTTP uses language tags within the Accept-Language, Content-Language, and URI-header fields.

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

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

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

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

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

where any two-letter primary-tag is an ISO 639 language abbreviation and any two-letter initial subtag is an ISO 3166 country code.

In the context of the Accept-Language header (Section 10.4), a language tag is not to be interpreted as a single token, as per RFC 1766, but as a hierarchy. A server should consider that it has a match when a language tag received in an Accept-Language header matches the initial portion of the language tag of a document. An exact match should be preferred. This interpretation allows a browser to send, for example:

```
Accept-Language: en-US, en; ql=0.95
```

when the intent is to access, in order of preference, documents in US-English ("en-US"), 'plain' or 'international' English ("en"), and any other variant of English (initial "en-").

> ***Note:*** *Using the language tag as a hierarchy does not imply that all languages with a common prefix will be understood by those fluent in one or more of those languages; it simply allows the user to request this commonality when it is true for that user.*

3.11 Logic Bags

A logic bag is a binary logic expression tree represented in prefix notation using the generic bag syntax. Logic bags are used by HTTP in the Unless (Section 10.40) header field as expressions to be tested against the requested resource's header field metainformation.

```
logic-bag      = "{" expression "}"

expression     = ( log-op 1*logic-bag )
               |  ( rel-op 1*field-tuple )
               |  ( "def" 1*field-name )

log-op         = "and" | "or" | "xor" | "not"
rel-op         = "eq" | "ne" | "lt" | "le" | "ge" | "gt" | "in"

field-tuple    = "{" field-name ( bag | token | quoted-string ) "}"
```

The recursive structure of a logic bag allows a complex expression tree to be formed by joining together subexpressions with logical operators. Expressions with relational operators are used to compare the requested resource's corresponding metainformation (header field values) to those inside the expression field-tuples. For example,

```
{or {ne {Content-MD5 "Q2hlY2sgSW50ZWdyaXR5IQ=="}}
    {ne {Content-Length 10036}}
    {ne {Content-Version "12.4.8"}}
    {gt {Last-Modified "Mon, 04 Dec 1995 01:23:45 GMT"}}}
```

The expression is evaluated recursively by depth-first traversal and bottom-up evaluation of the subexpressions until a true or false value can be determined. Multiple operands to an operator imply a conjunctive ("and") expression; e.g.,

```
{eq {A "a"} {B "b"} {C "c"}}
```

is equivalent to

```
{and {eq {A "a"}} {eq {B "b"}} {eq {C "c"}}}
```

Each expression is evaluated as defined by the operator:

and True if all of the operands evaluate true.

or True if any of the operands evaluate true.

xor True if one and only one operand evaluates true.

not True if all of the operands evaluate false.

eq True if all field-tuple values exactly match the resource's corresponding field values.

ne True if all field-tuple values do not match the resource's corresponding field values.

lt True if, for each field-tuple, the resource's corresponding field value is less than the one given in the expression.

le True if, for each field-tuple, the resource's corresponding field value is less than or equal to the one given in the expression.

ge True if, for each field-tuple, the resource's corresponding field value is greater than or equal to the one given in the expression.

gt True if, for each field-tuple, the resource's corresponding field value is greater than the one given in the expression.

in True if, for each field-tuple, the resource's corresponding field value contains the component value given in the expression.

def True if, for each field-name operand, the resource defines a value for that field.

A field-tuple consists of a field-name (assumed to be an HTTP header field name, though not constrained to those defined by this specification) and the field-value component which is to be compared against the resource's field value. The actual method of comparison (e.g., byte equivalence, substring matching, numeric order, substructure containment, etc.) is defined by the logical definition of the operator and the type of field-value allowed for that field-name. Server implementors must use an appropriate comparison function for each type of field-value given in this specification. The default functions for unrecognized fields are numeric comparison (for values consisting of 1*DIGIT) and lexical comparison (for all others).

Except for "ne", any comparison to a field not defined by the resource evaluates to false.

4. HTTP Message

4.1 Message Types

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

```
HTTP-message   = Simple-Request            ; HTTP/0.9 messages
               | Simple-Response
               | Full-Request              ; HTTP/1.1 messages
               | Full-Response
```

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

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

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

Simple-Request and Simple-Response do not allow the use of any header information and are limited to a single request method (GET).

```
Simple-Request  = "GET" SP Request-URI CRLF

Simple-Response = [ Entity-Body ]
```

Use of the Simple-Request format is discouraged because it prevents the client from using content negotiation and the server from identifying the media type of the returned entity.

4.2 Message Headers

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

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

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

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

The order in which header fields are received is not significant. However, it is "good practice" to send General-Header fields first, followed by Request-Header or Response-Header fields prior to the Entity-Header fields.

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

4.3 General Header Fields

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

```
General-Header  = Cache-Control          ; Section 10.8
                | Connection             ; Section 10.9
                | Date                   ; Section 10.17
                | Forwarded              ; Section 10.20
                | Keep-Alive             ; Section 10.24
                | MIME-Version           ; Section 10.28
                | Pragma                 ; Section 10.29
                | Upgrade                ; Section 10.41
```

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

5. Request

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

```
Request         = Simple-Request | Full-Request

Simple-Request  = "GET" SP Request-URI CRLF

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

If an HTTP/1.1 server receives a Simple-Request, it must respond with an HTTP/0.9 Simple-Response. An HTTP/1.1 client must never generate a Simple-Request.

5.1 Request-Line

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

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

Note that the difference between a Simple-Request and the Request-Line of a Full-Request is the presence of the HTTP-Version field and the availability of methods other than GET.

5.1.1 Method

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

```
Method          = "OPTIONS"                 ; Section 8.1
                | "GET"                     ; Section 8.2
                | "HEAD"                    ; Section 8.3
                | "POST"                    ; Section 8.4
                | "PUT"                     ; Section 8.5
                | "PATCH"                   ; Section 8.6
                | "COPY"                    ; Section 8.7
                | "MOVE"                    ; Section 8.8
                | "DELETE"                  ; Section 8.9
                | "LINK"                    ; Section 8.10
                | "UNLINK"                  ; Section 8.11
                | "TRACE"                   ; Section 8.12
                | "WRAPPED"                 ; Section 8.13
                | extension-method

extension-method = token
```

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

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

5.1.2 Request-URI

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

    Request-URI = "*" | absoluteURI | abs_path

The three options for Request-URI are dependent on the nature of the request. The asterisk "*" means that the request does not apply to a particular resource, but to the server itself, and is only allowed when the Method used does not necessarily apply to a resource. One example would be

```
OPTIONS * HTTP/1.1
```

The absoluteURI form is only allowed when the request is being made to a proxy. The proxy is requested to forward the request and return the response. If the request is GET or HEAD and a prior response is cached, the proxy may use the cached message if it passes any restrictions in the Cache-Control and Expires header fields. Note that the proxy may forward the request on to another proxy or directly to the server specified by the absoluteURI. In order to avoid request loops, a proxy must be able to recognize all of its server names, including any aliases, local variations, and the numeric IP address. An example Request-Line would be:

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

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

```
GET /pub/WWW/TheProject.html HTTP/1.1
```

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

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

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

would be forwarded by the proxy as

```
OPTIONS * HTTP/1.1
```

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

The Request-URI is transmitted as an encoded string, where some characters may be escaped using the "% hex hex" encoding defined by RFC 1738 [4]. The origin server must decode the Request-URI in order to properly interpret the request.

5.2 Request Header Fields

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

```
Request-Header  = Accept                    ; Section 10.1
                | Accept-Charset            ; Section 10.2
                | Accept-Encoding           ; Section 10.3
                | Accept-Language           ; Section 10.4
                | Authorization             ; Section 10.6
                | From                      ; Section 10.21
                | Host                      ; Section 10.22
                | If-Modified-Since         ; Section 10.23
                | Proxy-Authorization       ; Section 10.31
                | Range                     ; Section 10.33
                | Referer                   ; Section 10.34
                | Unless                    ; Section 10.40
                | User-Agent                ; Section 10.43
```

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 in the form of an HTTP response message.

```
Response         = Simple-Response | Full-Response

Simple-Response  = [ Entity-Body ]

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

A Simple-Response should only be sent in response to an HTTP/0.9 Simple-Request or if the server only supports the more limited HTTP/0.9 protocol. If a client sends an HTTP/1.1 Full-Request and receives a response that does not begin with a Status-Line, it should assume that the response is a Simple-Response and parse it accordingly. Note that the Simple-Response consists only of the entity body and is terminated by the server closing the connection.

6.1 Status-Line

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

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

Since a status line always begins with the protocol version and status code

```
"HTTP/" 1*DIGIT "." 1*DIGIT SP 3DIGIT SP
```

(e.g., `"HTTP/1.1 200 "`), the presence of that expression is sufficient to differentiate a Full-Response from a Simple-Response. Although the Simple-Response format may allow such an expression to occur at the beginning of an entity body, and thus cause a misinterpretation of the message if it was given in response to a Full-Request, most HTTP/0.9 servers are limited to responses of type "text/html" and therefore would never generate such a response.

6.1.1 Status Code and Reason Phrase

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

The first digit of the Status-Code defines the class of response. The last two digits do not have any categorization role. There are 5 values for the first digit:

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

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

```
Status-Code    = "100"   ; Continue
               | "101"   ; Switching Protocols
               | "200"   ; OK
               | "201"   ; Created
               | "202"   ; Accepted
               | "203"   ; Non-Authoritative Information
               | "204"   ; No Content
               | "205"   ; Reset Content
               | "206"   ; Partial Content
               | "300"   ; Multiple Choices
               | "301"   ; Moved Permanently
               | "302"   ; Moved Temporarily
               | "303"   ; See Other
               | "304"   ; Not Modified
               | "305"   ; Use Proxy
               | "400"   ; Bad Request
               | "401"   ; Unauthorized
               | "402"   ; Payment Required
               | "403"   ; Forbidden
               | "404"   ; Not Found
               | "405"   ; Method Not Allowed
               | "406"   ; None Acceptable
               | "407"   ; Proxy Authentication Required
               | "408"   ; Request Timeout
               | "409"   ; Conflict
               | "410"   ; Gone
               | "411"   ; Length Required
               | "412"   ; Unless True
               | "500"   ; Internal Server Error
               | "501"   ; Not Implemented
               | "502"   ; Bad Gateway
               | "503"   ; Service Unavailable
               | "504"   ; Gateway Timeout
               | extension-code

extension-code = 3DIGIT

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

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

6.2 Response Header Fields

The response header fields allow the server to pass additional information about the response which cannot be placed in the Status-Line. These header fields are not intended to give information about an Entity-Body returned in the response, but about access to the resource or the server itself.

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

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

7. Entity

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

7.1 Entity Header Fields

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

```
Entity-Header    = Allow                  ; Section 10.5
                 | Content-Encoding       ; Section 10.10
                 | Content-Language       ; Section 10.11
                 | Content-Length         ; Section 10.12
                 | Content-MD5            ; Section 10.13
                 | Content-Range          ; Section 10.14
                 | Content-Type           ; Section 10.15
                 | Content-Version        ; Section 10.16
                 | Derived-From           ; Section 10.18
                 | Expires                ; Section 10.19
                 | Last-Modified          ; Section 10.25
                 | Link                   ; Section 10.26
                 | Title                  ; Section 10.38
                 | Transfer-Encoding      ; Section 10.39
                 | URI-header             ; Section 10.42
                 | extension-header

extension-header = HTTP-header
```

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

7.2 Entity Body

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

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

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

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

7.2.1 Type

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

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

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

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

7.2.2 Length

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

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

Closing the connection cannot be used to indicate the end of a request body, since it leaves no possibility for the server to send back a response. For compatibility with HTTP/1.0 applications, HTTP/1.1 requests containing an entity body must include a valid Content-Length header field unless the server is known to be HTTP/1.1 compliant. HTTP/1.1 servers must accept the "chunked" transfer coding (Section 3.6) and multipart media types (Section 3.7.2), thus allowing either mechanism to be used for a request when Content-Length is unknown.

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

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

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

8. Method Definitions

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

The semantics of all methods may be affected by the presence of an Unless request header field, as described in Section 10.40.

8.1 OPTIONS

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

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

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

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

8.2 GET

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

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

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

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

8.3 HEAD

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

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

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

8.4 POST

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

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

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

HTTP/1.1 allows for a two-phase process to occur in accepting and processing a POST request. If the media type of the posted entity is not "application/x-www-form-urlencoded" [5], an HTTP/1.1 client must pause between sending the message header fields (including the empty line signifying the end of the headers) and sending the message body; the duration of the pause is five (5) seconds or until a response is received from the server, whichever is shorter. If no response is received during the pause period, or if the initial response is 100 (continue), the client may continue sending the POST request. If the response indicates an error, the client must discontinue the request and close the connection with the server after reading the response.

Upon receipt of a POST request, the server must examine the header fields and determine whether or not the client should continue its request. If any of the header fields indicate the request is insufficient or unacceptable to the server (i.e., will result in a 4xx or 5xx response), or if the server can determine the response without reading the entity body (e.g., a 301 or 302 response due to an old Request-URI), the server must send that response immediately upon its determination. If, on the other hand, the request appears (at least initially) to be acceptable and the client has indicated HTTP/1.1 compliance, the server must transmit an interim 100 response message after receiving the empty line terminating the request headers and continue processing the request. After processing has finished, a final response message must be sent to indicate the actual result of the request. A 100 response should not be sent in response to an HTTP/1.0 request except under experimental conditions, since an HTTP/1.0 client may mistake the 100 response for the final response.

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

The client can suggest one or more URIs for the new resource by including a URI header field in the request. However, the server should treat those URIs as advisory and may store the entity under a different URI, additional URIs, or without any URI.

The client may apply relationships between the new resource and other existing resources by including Link header fields, as described in Section 10.26. The server may use the Link information to perform other operations as a result of the new resource being added. For example, lists and indexes might be updated. However, no mandatory operation is imposed on the origin server. The origin server may also generate its own or additional links to other resources.

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

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

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

8.5 PUT

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

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

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

A single resource may be identified by many different URIs. For example, an article may have a URI for identifying "the current version" which is separate from the URI identifying each particular version. In this case, a PUT request on a general URI may result in several other URIs being defined by the origin server. The user agent should be informed of these URIs via one or more URI header fields in the response.

HTTP/1.1 allows for a two-phase process to occur in accepting and processing a PUT request. An HTTP/1.1 client must pause between sending the message header fields (including the empty line signifying the end of the headers) and sending the message body; the duration of the pause is five (5) seconds or until a response is received from the server, whichever is shorter. If no response is received during the pause period, or if the initial response is 100 (continue), the client may continue sending the PUT request. If the response indicates an error, the client must discontinue the request and close the connection with the server after reading the response.

Upon receipt of a PUT request, the server must examine the header fields and determine whether or not the client should continue its request. If any of the header fields indicate the request is insufficient or unacceptable to the server (i.e., will result in a 4xx or 5xx response), or if the server can determine the response without reading the entity body (e.g., a 301 or 302 response due to an old Request-URI), the server must send that response immediately upon its determination. If, on the other hand, the request appears (at least initially) to be acceptable and the client has indicated HTTP/1.1 compliance, the server must transmit an interim 100 response message after receiving the empty line terminating the request headers and continue processing the request. After processing has finished, a final response message must be sent to indicate the actual result of the request. A 100 response should not be sent in response to an HTTP/1.0 request except under experimental conditions, since an HTTP/1.0 client may mistake the 100 response for the final response.

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

The client can create or modify relationships between the enclosed entity and other existing resources by including Link header fields, as described in Section 10.26. As with POST, the server may use the Link information to perform other operations as a result of the request. However, no mandatory operation is imposed on the origin server. The origin server may generate its own or additional links to other resources.

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

8.6 PATCH

The PATCH method is similar to PUT except that the entity contains a list of differences between the original version of the resource identified by the Request-URI and the desired content of the resource after the PATCH action has been applied. The list of differences is in a format defined by the media type of the entity (e.g., "application/diff") and must include sufficient information to allow the server to recreate the changes necessary to convert the original version of the resource to the desired version.

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

HTTP/1.1 allows for a two-phase process to occur in accepting and processing a PATCH request. An HTTP/1.1 client must pause between sending the message header fields (including the empty line signifying the end of the headers) and sending the message body; the duration of the pause is five (5) seconds or until a response is received from the server, whichever is shorter. If no response is received during the pause period, or if the initial response is 100 (continue), the client may continue sending the PATCH request. If the response indicates an error, the client must discontinue the request and close the connection with the server after reading the response.

Upon receipt of a PATCH request, the server must examine the header fields and determine whether or not the client should continue its request. If any of the header fields indicate the request is insufficient or unacceptable to the server (i.e., will result in a 4xx or 5xx response), or if the server can determine the response without reading the entity body (e.g., a 301 or 302 response due to an old Request-URI), the server must send that response immediately upon its determination. If, on the other hand, the request appears (at least initially) to be acceptable and the client has indicated HTTP/1.1 compliance, the server must transmit an interim 100 response message after receiving the empty line terminating the request headers and continue processing the request. After processing has finished, a final response message must be sent to indicate the actual result of the request. A 100 response should not be sent in response to an HTTP/1.0 request except under experimental conditions, since an HTTP/1.0 client may mistake the 100 response for the final response.

For compatibility with HTTP/1.0 applications, all PATCH requests must include a valid Content-Length header field unless the server is known to be HTTP/1.1 compliant. When sending a PATCH request to an HTTP/1.1 server, a client must use at least one of: a valid Content-Length, a multipart Content-Type, or the "chunked" Transfer-Encoding. The server should respond with a 400 (bad request) message if it cannot determine the length of the request message's content, or with 411 (length required) if it wishes to insist on receiving a valid Content-Length.

The client can create or modify relationships between the new resource and other existing resources by including Link header fields, as described in Section 10.26. As with POST, the server may use the Link information to perform other operations as a result of the request. However, no mandatory operation is imposed on the origin server. The origin server may generate its own or additional links to other resources.

The actual method for determining how the patched resource is placed, and what happens to its predecessor, is defined entirely by the origin server. If version control is implemented by the origin server, then Link relationships should be defined by the server to help identify and control revisions to a resource. If the original version of the resource being patched included a Content-Version header field, the request entity must include a Derived-From header field corresponding to the value of the original Content-Version header field. Applications are encouraged to use these fields for constructing versioning relationships and resolving version conflicts.

8.7 COPY

The COPY method requests that the resource identified by the Request-URI be copied to the location(s) given in the URI header field of the request. Responses to this method are not cachable.

8.8 MOVE

The MOVE method requests that the resource identified by the Request-URI be moved to the location(s) given in the URI header field of the request. This method is equivalent to a COPY immediately followed by a DELETE, but enables both to occur within a single transaction.

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

8.9 DELETE

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

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

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

8.10 LINK

The LINK method establishes one or more Link relationships between the existing resource identified by the Request-URI and other existing resources. The difference between LINK and other methods allowing links to be established between resources is that the LINK method does not allow any Entity-Body to be sent in the request and does not directly result in the creation of new resources.

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

8.11 UNLINK

The UNLINK method removes one or more Link relationships from the existing resource identified by the Request-URI. These relationships may have been established using the LINK method or by any other method supporting the Link header. The removal of a link to a resource does not imply that the resource ceases to exist or becomes inaccessible for future references.

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

8.12 TRACE

The TRACE method requests that the server identified by the Request-URI reflect whatever is received back to the client as the entity body of the response. In this way, the client can see what is being received at the other end of the request chain, and may use this data for testing or diagnostic information.

If successful, the response should contain the entire, unedited request message in the entity body, with a Content-Type of "message/http", "application/http", or "text/plain". Responses to this method are not cachable.

8.13 WRAPPED

The WRAPPED method allows a client to send one or more encapsulated requests to the server identified by the Request-URI. This method is intended to allow the request(s) to be wrapped together, possibly encrypted in order to improve the security and/or privacy of the request, and delivered for unwrapping by the destination server. Upon receipt of the WRAPPED request, the destination server must unwrap the message and feed it to the appropriate protocol handler as if it were an incoming request stream.

Responses to this method are not cachable. Applications should not use this method for making requests that would normally be public and cachable.

The request entity must include at least one encapsulated message, with the media type identifying the protocol of that message. For example, if the wrapped request is another HTTP request message, then the media type must be either "message/http" (for a single message) or "application/http" (for a request stream containing one or more requests), with any codings identied by the Content-Encoding and Transfer-Encoding header fields.

HTTP/1.1 allows for a two-phase process to occur in accepting and processing a WRAPPED request. An HTTP/1.1 client must pause between sending the message header fields (including the empty line signifying the end of the headers) and sending the message body; the duration of the pause is five (5) seconds or until a response is received from the server, whichever is shorter. If no response is received during the pause period, or if the initial response is 100 (continue), the client may continue sending the WRAPPED request. If the response indicates an error, the client must discontinue the request and close the connection with the server after reading the response.

Upon receipt of a WRAPPED request, the server must examine the header fields and determine whether or not the client should continue its request. If any of the header fields indicate the request is insufficient or unacceptable to the server (i.e., will result in a 4xx or 5xx response), or if the server can determine the response without reading the entity body (e.g., a 301 or 302 response due to an old Request-URI), the server must send that response immediately upon its determination. If, on the other hand, the request appears (at least initially) to be acceptable and the client has indicated HTTP/1.1 compliance, the server must transmit an interim 100 response message after receiving the empty line terminating the request headers and continue processing the request. After processing has finished, a final response message must be sent to indicate the actual result of the request. A 100 response should not be sent in response to an HTTP/1.0 request except under experimental conditions, since an HTTP/1.0 client may mistake the 100 response for the final response.

For compatibility with HTTP/1.0 applications, all WRAPPED requests must include a valid Content-Length header field unless the server is known to be HTTP/1.1 compliant. When sending a WRAPPED request to an HTTP/1.1 server, a client must use at least one of: a valid Content-Length, a multipart Content-Type, or the "chunked" Transfer-Encoding. The server should respond with a 400 (bad request) message if it cannot determine the length of the request message's content, or with 411 (length required) if it wishes to insist on receiving a valid Content-Length.

9. Status Code Definitions

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

9.1 Informational 1xx

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

100 Continue

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

101 Switching Protocols

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

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

9.2 Successful 2xx

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

200 OK

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

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

HEAD the response must only contain the header information and no Entity-Body;

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

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

otherwise, an entity describing the result of the action;

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

201 Created

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

202 Accepted

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

The 202 response is intentionally non-committal. Its purpose is to allow a server to accept a request for some other process (perhaps a batch-oriented process that is only run once per day) without requiring that the user agent's connection to the server persist until the process is completed. The entity returned with this response should include an indication of the request's current status and either a pointer to a status monitor or some estimate of when the user can expect the request to be fulfilled.

203 Non-Authoritative Information

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

204 No Content

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

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

205 Reset Content

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

206 Partial Content

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

9.3 Redirection 3xx

This class of status code indicates that further action needs to be taken by the user agent in order to fulfill the request. The action required may be carried out by the user agent without interaction with the user if and only if the method used in the second request is GET or HEAD. A user agent should never automatically redirect a request more than 5 times, since such redirections usually indicate an infinite loop.

300 Multiple Choices

The requested resource is available at one or more locations and a preferred location could not be determined via preemptive content negotiation (Section 12). Unless it was a HEAD request, the response should include an entity containing a list of resource characteristics and locations from which the user or user agent can choose the one most appropriate. The entity format is specified by the media type given in the Content-Type header field. Depending upon the format and the capabilities of the user agent, selection of the most appropriate choice may be performed automatically. If the server has a preferred choice, it should include the URL in a Location field; user agents not capable of complex selection may use this field value for automatic redirection. This response is cachable unless indicated otherwise.

301 Moved Permanently

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

If the new URI is a single location, its URL must be given by the Location field in the response. If more than one URI exists for the resource, the primary URL should be given in the Location field and the other URIs given in one or more URI-header fields. Unless it was a HEAD request, the Entity-Body of the response should contain a short hypertext note with a hyperlink to the new URI(s).

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

302 Moved Temporarily

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

If the new URI is a single location, its URL must be given by the Location field in the response. If more than one URI exists for the resource, the primary URL should be given in the Location field and the other URIs given in one or more URI-header fields. Unless it was a HEAD request, the Entity-Body of the response should contain a short hypertext note with a hyperlink to the new URI(s).

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

303 See Other

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

If the new URI is a single location, its URL must be given by the Location field in the response. If more than one URI exists for the resource, the primary URL should be given in the Location field and the other URIs given in one or more URI-header fields. Unless it was a HEAD request, the Entity-Body of the response should contain a short hypertext note with a hyperlink to the new URI(s).

304 Not Modified

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

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

The 304 response must not include an entity body, and thus is always ternminated by the first empty line after the header fields.

305 Use Proxy

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

9.4 Client Error 4xx

The 4xx class of status code is intended for cases in which the client seems to have erred. If the client has not completed the request when a 4xx code is received, it should immediately cease sending data to the server. Except when responding to a HEAD request, the server should include an entity containing an explanation of the error situation, and whether it is a temporary or permanent condition. These status codes are applicable to any request method.

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

400 Bad Request

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

401 Unauthorized

The request requires user authentication. The response must include a WWW-Authenticate header field (Section 10.44) containing a challenge applicable to the requested resource. The client may repeat the request with a suitable Authorization header field (Section 10.6). If the request already included Authorization credentials, then the 401 response indicates that authorization has been refused for those credentials. If the 401 response contains the same challenge as the prior response, and the user agent has already attempted authentication at least once, then the user should be presented the entity that was given in the response, since that entity may include relevant diagnostic information. HTTP access authentication is explained in Section 11.

402 Payment Required

This code is reserved for future use.

403 Forbidden

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

404 Not Found

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

405 Method Not Allowed

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

406 None Acceptable

The server has found a resource matching the Request-URI, but not one that satisfies the conditions identified by the Accept and Accept-Encoding request headers. Unless it was a HEAD request, the response should include an entity containing a list of resource characteristics and locations from which the user or user agent can choose the one most appropriate. The entity format is specified by the media type given in the Content-Type header field. Depending upon the format and the capabilities of the user agent, selection of the most appropriate choice may be performed automatically.

407 Proxy Authentication Required

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

408 Request Timeout

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

409 Conflict

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

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

410 Gone

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

The 410 response is primarily intended to assist the task of web maintenance by notifying the recipient that the resource is intentionally unavailable and that the server owners desire that remote links to that resource be removed. Such an event is common for limited-time, promotional services and for resources belonging to individuals no longer working at the server's site. It is not necessary to mark all permanently unavailable resources as "gone" or to keep the mark for any length of time — that is left to the discretion of the server owner.

411 Length Required

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

412 Unless True

The condition given in the Unless request-header field (Section 10.40) evaluated to true when it was tested on the server and the request did not include a Range header field (which would indicate a partial GET) or an If-Modified-Since header field (which would indicate a conditional GET). This response code allows the client to place arbitrary preconditions on the current resource metainformation (header field data) and thus prevent the requested method from being applied to a resource other than the one intended.

9.5 Server Error 5xx

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

500 Internal Server Error

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

501 Not Implemented

The server does not support the functionality required to fulfill the request. This is the appropriate response when the server does not recognize the request method and is not capable of supporting it for any resource.

502 Bad Gateway

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

503 Service Unavailable

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

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

504 Gateway Timeout

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

10. Header Field Definitions

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

10.1 Accept

The Accept response-header field can be used to indicate a list of media ranges which are acceptable as a response to the request. The asterisk "*" character is used to group media types into ranges, with "*/*" indicating all media types and "type/*" indicating all subtypes of that type. The set of ranges given by the client should represent what types are acceptable given the context of the request. The Accept field should only be used when the request is specifically limited to a set of desired types, as in the case of a request for an in-line image, or to indicate qualitative preferences for specific media types.

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

```
Accept          = "Accept" ":" #(
                       media-range
                       [ ";" "q" "=" qvalue ]
                       [ ";" "mxb" "=" 1*DIGIT ] )
media-range     = ( "*/*"
                | ( type "/" "*" )
                | ( type "/" subtype )
                  ) *( ";" parameter )
```

The parameter q is used to indicate the quality factor, which represents the user's preference for that range of media types. The parameter mxb gives the maximum acceptable size of the Entity-Body, in decimal number of octets, for that range of media types. Section 12 describes the content negotiation algorithm which makes use of these values. The default values are: q=1 and mxb=undefined (i.e., infinity).

The example

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

should be interpreted as "I prefer audio/basic, but send me any audio type if it is the best available after an 80% mark-down in quality."

If no Accept header is present, then it is assumed that the client accepts *all* media types with quality factor 1. This is equivalent to the client sending the following accept header field:

```
Accept: */*; q=1
```

or

```
Accept: */*
```

If a single Accept header is provided and it contains no field value, then the server must interpret it as a request to not perform any preemptive content negotiation (Section 12) and instead return a 406 (none acceptable) response if there are variants available for the Request-URI.

A more elaborate example is

```
Accept: text/plain; q=0.5, text/html,
        text/x-dvi; q=0.8; mxb=100000, text/x-c
```

Verbally, this would be interpreted as “text/html and text/x-c are the preferred media types, but if they do not exist, then send the text/x-dvi entity if it is less than 100000 bytes, otherwise send the text/plain entity.”

Media ranges can be overridden by more specific media ranges or specific media types. If more than one media range applies to a given type, the most specific reference has precedence. For example,

```
Accept: text/*, text/html, text/html;version=2.0, */*
```

have the following precedence:

```
1) text/html;version=2.0
2) text/html
3) text/*
4) */*
```

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

```
Accept: text/*;q=0.3, text/html;q=0.7, text/html;version=2.0,
        */*;q=0.5
```

would cause the following values to be associated:

```
text/html;version=2.0         = 1
text/html                     = 0.7
text/plain                    = 0.3
image/jpeg                    = 0.5
text/html;level=3             = 0.7
```

It must be emphasized that the Accept field should only be used when it is necessary to restrict the response media types to a subset of those possible or when the user has been permitted to specify qualitative values for ranges of media types. If no quality factors have been set by the user, and the context of the request is such that the user agent is capable of saving the entity to a file if the received media type is unknown, then the only appropriate value for Accept is "*/*", or an empty value if the user desires reactive negotiation.

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

10.2 Accept-Charset

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

```
Accept-Charset    = "Accept-Charset" ":" 1#charset
```

Character set values are described in Section 3.4. An example is

```
Accept-Charset: iso-8859-1, unicode-1-1
```

If no Accept-Charset field is given, the default is that any character set is acceptable. If the Accept-Charset field is given and the requested resource is not available in one of the listed character sets, then the server should respond with the 406 (none acceptable) status code.

10.3 Accept-Encoding

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

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

An example of its use is

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

If no Accept-Encoding field is present in a request, the server may assume that the client will accept any content coding. If an Accept-Encoding field is present, but contains an empty field value, then the user agent is refusing to accept any content coding.

10.4 Accept-Language

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

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

The language-tag is described in Section 3.10. Each language may be given an associated quality value which represents an estimate of the user's comprehension of that language. The quality value defaults to "q=1" (100% comprehension) for listed languages. This value may be used in the server's content negotiation algorithm (Section 12). For example,

```
Accept-Language: da, en-gb;q=0.8, de;q=0.55
```

would mean: “I prefer Danish, but will accept British English (with 80% comprehension) or German (with a 55% comprehension).”

If the server cannot fulfill the request with one or more of the languages given, or if the languages only represent a subset of a multi-linguistic Entity-Body, it is acceptable to serve the request in an unspecified language. This is equivalent to assigning a quality value of "q=0.001" to any unlisted language.

If no Accept-Language header is present in the request, the server should assume that all languages are equally acceptable.

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

10.5 Allow

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

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

Example of use:

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

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

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

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

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

10.6 Authorization

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

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

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

Responses to requests containing an Authorization field are not cachable.

10.7 Base

The Base entity-header field may be used to specify the base URI for resolving relative URLs, as described in RFC 1808 [11].

10.8 Cache-Control

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

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

cache-directive   = "cachable"
                  | "max-age" "=" delta-seconds
                  | "private" [ "=" <"> 1#field-name <"> ]
                  | "no-cache" [ "=" <"> 1#field-name <"> ]
```

The Cache-Control header field may be used to modify the optional behavior of caching mechanisms, and the default cachability of a response message; it cannot be used to modify the required behavior of caching mechanisms. HTTP requirements for caching and cachable messages are described in Section 13.

The "cachable" directive indicates that the entire response message is cachable unless required otherwise by HTTP restrictions on the request method and response code. In other words, this directive indicates that the server believes the response to be cachable. This directive applies only to responses and must not be used with any other cache directive.

When the "`max-age`" directive is present in a request message, an application must forward the request toward the origin server if it has no cached copy, or refresh its cached copy if it is older than the age value given (in seconds) prior to returning a response. A cached copy's age is determined by the cached message's Date header field, or the equivalent as stored by the cache manager.

In most cases, a cached copy can be refreshed by forwarding a conditional GET request toward the origin server with the stored message's Last-Modified value in the If-Modified-Since field. The Unless header field may be used to add further restrictions to the modification test on the server. If a 304 (not modified) response is received, the cache should replace the cached message's Date with that of the 304 response and send this refreshed message as the response. Any other response should be forwarded directly to the requestor and, depending on the response code and the discretion of the cache manager, may replace the message in the cache.

When the "`max-age`" directive is present in a cached response message, an application must refresh the message if it is older than the age value given (in seconds) at the time of a new request for that resource. The behavior should be equivalent to what would occur if the request had included the max-age directive. If both the new request and the cached message have max-age specified, then the lesser of the two values must be used. A max-age value of zero (0) forces a cache to perform a refresh (If-Modified-Since) on every request. The max-age directive on a response implies that the server believes it to be cachable.

The "`private`" directive indicates that parts of the response message are intended for a single user and must not be cached except within a private (non-shared) cache controlled by the user agent. If no list of field names is given, then the entire message is private; otherwise, only the information within the header fields identified by the list of names is private and the remainder of the message is believed to be cachable by any application. This allows an origin server to state that the specified parts of the message are intended for only one user and are not a valid response for requests by other agents. The "private" directive is only applicable to responses and must not be generated by clients.

> ***Note:*** *This usage of the word "private" implies only that the message must not be cached publically; it does not ensure the privacy of the message content.*

The “`no-cache`” directive on a request message requires any cache to forward the request toward the origin server even if it has a cached copy of what is being requested. This allows a client to insist upon receiving an authoritative response to its request. It also allows a client to refresh a cached copy which is known to be corrupted or stale. This is equivalent to the "no-cache" pragma-directive in Section 10.29. The list of field names is not used with requests and must not be generated by clients. The no-cache directive overrides any max-age directive.

The “`no-cache`” directive on a response message indicates that parts of the message must never be cached. If no list of field names is given, then the entire message must not be cached; otherwise, only the information within the header fields identified by the list of names must not be cached and the remainder of the message is believed to be cachable. This allows an origin server to state that the specified parts of the message are intended for only one recipient and must not be stored unless the user explicitly requests it through a separate action.

The max-age, private, and no-cache directives may be used in combination to define the cachability of each part of the message. In all cases, no-cache takes precedence over private, which in turn takes precedence over max-age.

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

10.9 Connection

The Connection general-header field is used to indicate a list of keywords and header field names containing information which is only applicable to the current connection between the sender and the nearest non-tunnel recipient on the request/response chain. This information must not be forwarded or cached. Unlike the default behavior, the recipient cannot safely ignore the semantics of the listed field-names if they are not understood, since forwarding them may imply that understanding.

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

Proxies and gateways must discard the named header fields, and the Connection header itself, before forwarding the message. Proxies and gateways may add their own Connection information to forwarded messages if such options are desired for the forwarding connection. These restrictions do not apply to a tunnel, since the tunnel is acting as a relay between two connections and does not affect the connection options.

Whether or not the listed field-name(s) occur as header fields in the message is optional. If no corresponding header field is present, then the field name is treated as a keyword. Keywords are useful for indicating a desired option without assigning parameters to that option. This allows for a minimal syntax to provide connection-based options without pre-restricting the syntax or number of those options. HTTP/1.1 only defines the "keep-alive" keyword.

The semantics of Connection are defined by HTTP/1.1 in order to provide a safe transition to connection-based features. Connection header fields received in an HTTP/1.0 message, as would be the case if an older proxy mistakenly forwards the field, cannot be trusted and must be discarded except under experimental conditions.

10.9.1 Persistent Connections

The "keep-alive" keyword in a Connection header field allows the sender to indicate its desire for a persistent connection (i.e., a connection that lasts beyond the current request/response transaction). Persistent connections allow the client to perform multiple requests without the overhead of connection tear-down and set-up between each request.

As an example, a client would send

```
Connection: Keep-Alive
```

to indicate that it desires to keep the connection open for multiple requests. The server may then respond with a message containing

```
Connection: Keep-Alive
```

to indicate that the connection will be kept open for the next request. The Connection header field with a keep-alive keyword must be sent on all requests and responses that wish to continue the persistence. The client sends requests as normal and the server responds as normal, except that all messages containing an entity body must have a length that can be determined without closing the connection (i.e., each message containg an entity body must have a valid Content-Length, be a multipart media type, or be encoded using the "chunked" transfer coding, as described in Section 7.2.2).

The Keep-Alive header field (Section 10.24) may be used to include diagnostic information and other optional parameters. For example, the server may responds with

```
Connection: Keep-Alive
Keep-Alive: timeout=10, max=5
```

to indicate that the server has selected (perhaps dynamically) a maximum of 5 requests, but will timeout if the next request is not received within 10 seconds. Note, however, that this additional information is optional and the Keep-Alive header field does not need to be present. If it is present, the semantics of the Connection header field prevents it from being accidentally forwarded to downstream connections.

The persistent connection ends when either side closes the connection or after the receipt of a response which lacks the "keep-alive" keyword. The server may close the connection immediately after responding to a request without a "keep-alive" keyword. A client can tell if the connection will be closed by looking for a "keep-alive" in the response.

10.10 Content-Encoding

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

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

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

```
Content-Encoding: gzip
```

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

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

10.11 Content-Language

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

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

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

```
Content-Language: dk
```

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

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

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

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

Content-Language may be applied to any media type — it should not be limited to textual documents.

10.12 Content-Length

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

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

An example is

```
Content-Length: 3495
```

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

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

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

10.13 Content-MD5

TBS

10.14 Content-Range

TBS

10.15 Content-Type

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

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

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

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

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

10.16 Content-Version

The Content-Version entity-header field defines the version tag associated with a rendition of an evolving entity. Together with the Derived-From field described in Section 10.18, it allows a group of people to work simultaneously on the creation of a work as an iterative process. The field should be used to allow evolution of a particular work along a single path. It should not be used to indicate derived works or renditions in different representations. It may also me used as an opaque value for comparing a cached entity's version with that of the current resource.

```
Content-Version   = "Content-Version" ":" quoted-string
```

Examples of the Content-Version field include:

```
Content-Version: "2.1.2"
Content-Version: "Fred 19950116-12:26:48"
Content-Version: "2.5a4-omega7"
```

The value of the Content-Version field should be considered opaque to all parties but the origin server. A user agent may suggest a value for the version of an entity transferred via a PUT request; however, only the origin server can reliably assign that value.

10.17 Date

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

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

An example is

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

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

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

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

10.18 Derived-From

The Derived-From entity-header field can be used to indicate the version tag of the resource from which the enclosed entity was derived before modifications were made by the sender. This field is used to help manage the process of merging successive changes to a resource, particularly when such changes are being made in parallel and from multiple sources.

```
Derived-From    = "Derived-From" ":" quoted-string
```

An example use of the field is:

```
Derived-From: "2.1.1"
```

The Derived-From field is required for PUT and PATCH requests if the entity being sent was previously retrieved from the same URI and a Content-Version header was included with the entity when it was last retrieved.

10.19 Expires

The Expires entity-header field gives the date/time after which the entity should be considered stale. This allows information providers to suggest the volatility of the resource, or a date after which the information may no longer be valid. Applications must not cache this entity beyond the date given. The presence of an Expires field does not imply that the original resource will change or cease to exist at, before, or after that time. However, information providers that know or even suspect that a resource will change by a certain date should include an Expires header with that date. The format is an absolute date and time as defined by HTTP-date in Section 3.3.

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

An example of its use is

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

If the date given is equal to or earlier than the value of the Date header, the recipient must not cache the enclosed entity. If a resource is dynamic by nature, as is the case with many data-producing processes, entities from that resource should be given an appropriate Expires value which reflects that dynamism.

The Expires field cannot be used to force a user agent to refresh its display or reload a resource; its semantics apply only to caching mechanisms, and such mechanisms need only check a resource's expiration status when a new request for that resource is initiated.

User agents often have history mechanisms, such as "Back" buttons and history lists, which can be used to redisplay an entity retrieved earlier in a session. By default, the Expires field does not apply to history mechanisms. If the entity is still in storage, a history mechanism should display it even if the entity has expired, unless the user has specifically configured the agent to refresh expired history documents.

> ***Note:*** *Applications are encouraged to be tolerant of bad or misinformed implementations of the Expires header. A value of zero (0) or an invalid date format should be considered equivalent to an "expires immediately." Although these values are not legitimate for HTTP/1.1, a robust implementation is always desirable.*

10.20 Forwarded

The Forwarded general-header field is to be used by gateways and proxies to indicate the intermediate steps between the user agent and the server on requests, and between the origin server and the client on responses. It is analogous to the "Received" field of RFC 822 [9] and is intended to be used for tracing transport problems and avoiding request loops.

```
Forwarded      = "Forwarded" ":" #( "by" URI [ "(" product ")" ]
                   [ "for" FQDN ] )

FQDN           = <Fully-Qualified Domain Name>
```

For example, a message could be sent from a client on *ptsun00.cern.ch* to a server at *www.ics.uci.edu* port 80, via an intermediate HTTP proxy at *info.cern.ch* port 8000. The request received by the server at *www.ics.uci.edu* would then have the following Forwarded header field:

```
Forwarded: by http://info.cern.ch:8000/ for ptsun00.cern.ch
```

Multiple Forwarded header fields are allowed and should represent each proxy/gateway that has forwarded the message. It is strongly recommended that proxies/gateways used as a portal through a network firewall do *not*, by default, send out information about the internal hosts within the firewall region. This information should only be propagated if explicitly enabled. If not enabled, the for token and FQDN should not be included in the field value, and any Forwarded headers already present in the message (those added behind the firewall) should be removed.

10.21 From

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

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

An example is:

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

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

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

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

10.22 Host

The Host request-header field allows the client to specify, for the server's benefit, the Internet host given by the original Uniform Resource Identifier (Section 3.2) of the resource being requested, as it was obtained from the user or the referring resource. This allows a server to differentiate between internally-ambiguous URLs (such as the root "/" URL of a server harboring multiple virtual hostnames). This field is required on all HTTP/1.1 requests which do not already include the host in the Request-URI.

```
Host           = "Host" ":" host            ; Section 3.2.2
```

Example:

```
Host: www.w3.org
```

The contents of the Host header field should exactly match the host information used to contact the origin server or gateway in question. It must not include the trailing ":port" information which may also be found in the net_loc portion of a URL (Section 3.2).

10.23 If-Modified-Since

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

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

An example of the field is:

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

A conditional GET method requests that the identified resource be transferred only if it has been modified since the date given by the If-Modified-Since header. The algorithm for determining this includes the following cases:

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

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

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

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

10.24 Keep-Alive

The Keep-Alive general-header field may be used to include diagnostic information and other optional parameters associated with the "keep-alive" keyword of the Connection header field (Section 10.9). This Keep-Alive field must only be used when the "keep-alive" keyword is present (Section 10.9.1).

```
Keep-Alive      = "Keep-Alive" ":" 1#kaparam
kaparam         = ( "timeout" "=" delta-seconds )
                | ( "max" "=" 1*DIGIT )
                | ( attribute [ "=" value ] )
```

The Keep-Alive header field and the additional information it provides are optional and do not need to be present to indicate a persistent connection has been established. The semantics of the Connection header field prevent the Keep-Alive field from being accidentally forwarded to downstream connections.

HTTP/1.1 defines semantics for the optional "timeout" and "max" parameters on responses; other parameters may be added and the field may also be used on request messages. The "timeout" parameter allows the server to indicate, for diagnostic purposes only, the amount of time in seconds it is currently allowing between when the response was generated and when the next request is received from the client (i.e., the request timeout limit). Similarly, the "max" parameter allows the server to indicate the maximum additional requests that it will allow on the current persistent connection.

For example, the server may respond to a request for a persistent connection with

```
Connection: Keep-Alive
Keep-Alive: timeout=10, max=5
```

to indicate that the server has selected (perhaps dynamically) a maximum of 5 requests, but will timeout the connection if the next request is not received within 10 seconds. Although these parameters have no affect on the operational requirements of the connection, they are sometimes useful for testing functionality and monitoring server behavior.

10.25 Last-Modified

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

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

An example of its use is

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

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

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

10.26 Link

The Link entity-header field provides a means for describing a relationship between the entity and some other resource. An entity may include multiple Link values. Links at the metainformation level typically indicate relationships like hierarchical structure and navigation paths. The Link field is semantically equivalent to the <LINK> element in HTML [5].

```
Link             = "Link" ":" #("<" URI ">"
                   [ ";" "rel" "=" relationship ]
                   [ ";" "rev" "=" relationship ]
                   [ ";" "title" "=" quoted-string ] )

relationship     = sgml-name
                 | ( <"> sgml-name *( SP sgml-name) <"> )

sgml-name        = ALPHA *( ALPHA | DIGIT | "." | "-" )
```

Relationship values are case-insensitive and may be extended within the constraints of the sgml-name syntax. The title parameter may be used to label the destination of a link such that it can be used as identification within a human-readable menu.

Examples of usage include:

```
Link: <http://www.cern.ch/TheBook/chapter2>; rel="Previous"
Link: <mailto:timbl@w3.org>; rev="Made"; title="Tim Berners-Lee"
```

The first example indicates that chapter2 is previous to the entity in a logical navigation path. The second indicates that the person responsible for making the resource available is identified by the given e-mail address.

10.27 Location

The Location response-header field defines the exact location of the resource that was identified by the Request-URI. For 2xx responses, if the Request-URI corresponds to a negotiable set of variants and the response includes one of those variants, then the response must also include a Location header field containing the exact location of the chosen variant. For 3xx responses, the location should indicate the server's preferred URL for automatic redirection to the resource. The field value consists of a single absolute URL.

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

An example is

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

If no base URL is provided by or within the entity, the value of the Location field should be used as the base for resolving relative URLs [11].

10.28 MIME-Version

HTTP is not a MIME-compliant protocol (see Appendix C). However, HTTP/1.1 messages may include a single MIME-Version general-header field to indicate what version of the MIME protocol was used to construct the message. Use of the MIME-Version header field indicates that the message is in full compliance with the MIME protocol (as defined in [7]). Proxies/gateways are responsible for ensuring full compliance (where possible) when exporting HTTP messages to strict MIME environments.

```
MIME-Version    = "MIME-Version" ":" 1*DIGIT "." 1*DIGIT
```

MIME version "`1.0`" is the default for use in HTTP/1.1. However, HTTP/1.1 message parsing and semantics are defined by this document and not the MIME specification.

10.29 Pragma

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

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

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

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

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

10.30 Proxy-Authenticate

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

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

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

10.31 Proxy-Authorization

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

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

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

10.32 Public

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

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

Example of use:

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

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

10.33 Range

TBS

10.34 Referer

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

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

Example:

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

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

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

10.35 Refresh

TBS

10.36 Retry-After

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

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

Two examples of its use are

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

In the latter example, the delay is 2 minutes.

10.37 Server

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

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

Example:

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

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

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

10.38 Title

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

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

An example of the field is

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

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

10.39 Transfer Encoding

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

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

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

```
Transfer-Encoding: chunked
```

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

10.40 Unless

The Unless request-header field performs a similar function as If-Modified-Since, but the comparison is based on any Entity-Header field value of the resource and is not restricted to the GET method.

```
Unless               = "Unless" ":" 1#logic-bag
```

For example,

```
Unless: {or {ne {Content-MD5 "Q2hlY2sgSW50ZWdyaXR5IQ=="}}
            {ne {Content-Length 10036}}
            {ne {Content-Version "12.4.8"}}
            {gt {Last-Modified "Mon, 04 Dec 1995 01:23:45 GMT"}}}
```

Multiple Unless headers, or multiple bags separated by commas, can be combined by OR’ing them together:

```
Unless: {eq {A "a"}}
Unless: {eq {B "b"}}
```

is equivalent to

```
Unless: {eq {A "a"}},{eq {B "b"}}
```

which in turn is equivalent to

```
Unless: {or {eq {A "a"}} {eq {B "b"}}}
```

When a request containing an Unless header field is received, the server must evaluate the expression defined by the listed logic-bags (Section 3.11). If the expression evaluates to false, then no change is made to the semantics of the request. If it evaluates true and the request is not a conditional GET (If-Modified-Since, Section 10.23) or a partial GET (Range, Section 10.33), then the server must abort the request and respond with the 412 (unless true) status code. If the request is a conditional GET, then the server must disregard the If-Modified-Since value and respond as it would for a normal GET. Similarly, if the request is a partial GET, then the server must disregard the Range value and respond as it would for a normal GET.

10.41 Upgrade

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

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

For example,

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

The purpose of the Upgrade header is to allow easier migration across protocols in order to better match the application needs with protocol capabilities.

10.42 URI

The URI entity-header field is used to inform the recipient of other Uniform Resource Identifiers (Section 3.2) by which the resource can be identified, and of all negotiable variants corresponding to the Request-URI.

```
URI-header     = "URI" ":" 1#( uri-mirror | uri-name | uri-variant )

uri-mirror     = "{" "mirror" <"> URI <"> "}"
uri-name       = "{" "name" <"> URI <"> "}"
uri-variant    = "{" "variant" <"> URI <"> qvalue
                     [ "{" "type" <"> media-type <"> "}" ]
                     [ "{" "language" <"> 1#language-tag <"> "}" ]
                     [ "{" "encoding" <"> 1#content-coding <"> "}" ]
                     [ "{" "length" 1*DIGIT "}" ]
                     [ "{" "user-agent" "}" ]
                 "}"
```

Any URI specified in this field can be absolute or relative to the Request-URI. The "mirror" form of URI refers to a location which is a mirror copy of the Request-URI. The "name" form refers to a location-independent name corresponding to the Request-URI. The "variant" form refers to one of the set of negotiable variants that may be retrieved via a request on the Request-URI.

If the Request-URI maps to a set of variants, then the dimensions of that variance must be given in any response containing one of those variants. If the Location header field is present in a 2xx response, its value identifies which one of the variants is included with the response. An example is:

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

URI: {variant "TheProject.fr.html" 1.0
             {type "text/html"} {language "fr"}},
     {variant "TheProject.en.html" 1.0
             {type "text/html"} {language "en"}},
     {variant "TheProject.fr.txt" 0.7
             {type "text/plain"} {language "fr"}},
     {variant "TheProject.en.txt" 0.8
             {type "text/plain"} {language "en"}}
```

which indicates that the negotiable Request-URI covers a group of four individual resources that vary in media type and natural language. The type, language, encoding, and length attributes refer to their Content-* counterparts for each resource. The user-agent attribute indicates that the associated URI is negotiable based on the User-Agent header field.

User agents may use this information to notify the user of additional formats and to guide the process of reactive content negotiation (Section 12).

10.43 User-Agent

The User-Agent request-header field contains information about the user agent originating the request. This is for statistical purposes, the tracing of protocol violations, and automated recognition of user agents for the sake of tailoring responses to avoid particular user agent limitations. Although it is not required, user agents should include this field with requests. The field can contain multiple product tokens (Section 3.8) and comments identifying the agent and any subproducts which form a significant part of the user agent. By convention, the product tokens are listed in order of their significance for identifying the application.

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

Example:

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

10.44 WWW-Authenticate

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

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

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

11. Access Authentication

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

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

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

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

The realm attribute (case-insensitive) is required for all authentication schemes which issue a challenge. The realm value (case-sensitive), in combination with the canonical root URL of the server being accessed, defines the protection space. These realms allow the protected resources on a server to be partitioned into a set of protection spaces, each with its own authentication scheme and/or authorization database. The realm value is a string, generally assigned by the origin server, which may have additional semantics specific to the authentication scheme.

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

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

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

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

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

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

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

11.1 Basic Authentication Scheme

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

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

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

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

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

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

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

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

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

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

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

11.2 Digest Authentication Scheme

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

12. Content Negotiation

Content negotiation is an optional feature of the HTTP protocol. It is designed to allow for selection of a preferred content representation based upon the attributes of the negotiable variants corresponding to the requested resource. HTTP/1.1 provides for two types of negotiation: preemptive and reactive.

Servers that make use of content negotiated resources must include URI response headers which accurately describe the available variants, and include the relevant parameters necessary for the client (user agent or proxy) to evaluate those variants.

12.1 Preemptive Negotiation

Preemptive negotiation attempts to "negotiate" the variant parameters by including the user agent preferences within each request. In this way, the preferred representation of the resource may be negotiated and obtained within a single request-response round-trip, and without intervention from the user. However, this also means that the user agent preferences are all the time, even though relatively few resources are ever negotiable. Preemptive negotiation may not always be desirable for the user and is sometimes unnecessary for the content provider. Implementors should provide mechanisms whereby the amount of preemptive content negotiation, and the parameters of that negotiation, are configurable by the user and server maintainer.

The first step in the negotiation algorithm is for the server to determine whether or not there are any content variants for the requested resource. Content variants may be in the form of multiple preexisting entities or a set of dynamic conversion filters. These variants make up the set of entities which may be sent in response to a request for the given Request-URI. In most cases, there will only be one available form of the resource, and thus a single "variant".

For each variant form of the resource, the server identifies a set of quality values (Section 3.9) which act as weights for measuring the desirability of that resource as a response to the current request. The calculated weights are all real numbers in the range 0 through 1, where 0 is the minimum and 1 the maximum value. The maximum acceptable bytes for each media range and the size of the resource variant are also factors in the equation.

The following parameters are included in the calculation:

qs Source quality is measured by the content provider as representing the amount of degradation from the original source. For example, a picture originally in JPEG form would have a lower qs when translated to the XBM format, and much lower qs when translated to an ASCII-art representation. Note, however, that this is a function of the source — an original piece of ASCII-art may degrade in quality if it is captured in JPEG form. The qs value should be assigned to each variant by the content provider; if no qs value has been assigned, the default is generally "qs=1". A server may define its own default qs value based on the resource characteristics, but only if individual resources can override those defaults.

qe Encoding quality is measured by comparing the variant's applied content-codings (Section 3.5) to those listed in the request message's Accept-Encoding field. If the variant has no assigned Content-Encoding, or if no Accept-Encoding field is present, the value assigned is "qe=1". If all of the variant's content encodings are listed in the Accept-Encoding field, then the value assigned is "qe=1". If any of the variant's content encodings are not listed in the provided Accept-Encoding field, then the value assigned is "qe=0".

qc Charset quality is measured by comparing the variant media-type's charset parameter value (if any) to those character sets (Section 3.4) listed in the request message's Accept-Charset field. If the variant's media-type has no charset parameter, or the variant's charset is US-ASCII, or if no Accept-Charset field is present, then the value assigned is "qc=1". If the variant's charset is listed in the Accept-Charset field, then the value assigned is "qc=1". Otherwise, if the variant's charset is not listed in the provided Accept-Encoding field, then the value assigned is "qc=0".

ql Language quality is measured by comparing the variant's assigned language tag(s) (Section 3.10) to those listed in the request message's Accept-Language field. If no variant has an assigned Content-Language, or if no Accept-Language field is present, the value assigned is "ql=1". If at least one variant has an assigned content language, but the one currently under consideration does not, then it should be assigned the value "ql=0.5". If any of the variant's content languages are listed in the Accept-Language field, then the value assigned is the maximum of the "q" parameter values for those language tags (Section 10.4); if there was no exact match and at least one of the Accept-Language field values is a complete subtag prefix of the content language tag(s), then the "q" parameter value of the largest matching prefix is used. If none of the variant's content language tags or tag prefixes are listed in the provided Accept-Language field, then the value assigned is "ql=0.001".

q Media type quality is measured by comparing the variant's assigned media type (Section 3.7) to those listed in the request message's Accept field. If no Accept field is given, then the value assigned is "q=1". If at least one listed media range (Section 10.1) matches the variant's media type, then the "q" parameter value assigned to the most specific of those matched is used (e.g., "text/html;version=3.0" is more specific than "text/html", which is more specific than "text/*", which in turn is more specific than "*/*"). If no media range in the provided Accept field matches the variant's media type, then the value assigned is "q=0".

mxb The maximum number of bytes in an Entity-Body that the client will accept is also obtained from the matching of the variant's assigned media type to those listed in the request message's Accept field. If no Accept field is given, or if no media range in the provided Accept field matches the variant's media type, then the value assigned is "mxb=undefined" (i.e., infinity). Otherwise, the value used is that given to the "mxb" parameter in the media range chosen above for the q value.

bs The actual number of bytes in the Entity-Body for the variant when it is included in a response message. This should equal the value of Content-Length.

The mapping function is defined as:

```
Q(qs,qe,qc,ql, { if mxb=undefined, then (qs*qe*qc*ql*q) }
  q,mxb,bs)    = { if mxb >= bs,     then (qs*qe*qc*ql*q) }
                 { if mxb <  bs,     then 0               }
```

The variants with a maximal value for the Q function represent the preferred representation(s) of the entity; those with a Q values less than the maximal value are therefore excluded from further consideration. If multiple representations exist that only vary by Content-Encoding, then the smallest representation (lowest bs) is preferred.

If no variants remain with a value of Q greater than zero (0), the server should respond with a 406 (none acceptable) response message. If multiple variants remain with an equally high Q value, the server may either choose one from those available and respond with 200 (ok) or respond with 300 (multiple choices) and include an entity describing the choices. In the latter case, the entity should either be of type "text/html', such that the user can choose from among the choices by following an exact link, or of some type that would allow the user agent to perform the selection automatically.

The 300 (multiple choices) response can be given even if the server does not perform any winnowing of the representation choices via the content negotiation algorithm described above. Furthermore, it may include choices that were not considered as part of the negotiation algorithm and resources that may be located at other servers.

The algorithm presented above assumes that the user agent has correctly implemented the protocol and is accurately communicating its intentions in the form of Accept-related header fields. The server may alter its response if it knows that the particular version of user agent software making the request has incorrectly or inadequately implemented these fields.

13. Caching

[This will be a summary of what is already defined in the Methods, Status Codes, Cache-Control, Unless, and If-Modified-Since sections.]

14. Security Considerations

This section is meant to inform application developers, information providers, and users of the security limitations in HTTP/1.1 as described by this document. The discussion does not include definitive solutions to the problems revealed, though it does make some suggestions for reducing security risks.

14.1 Authentication of Clients

As mentioned in Section 11.1, the Basic authentication scheme is not a secure method of user authentication, nor does it prevent the Entity-Body from being transmitted in clear text across the physical network used as the carrier. HTTP does not prevent additional authentication schemes and encryption mechanisms from being employed to increase security.

14.2 Safe Methods

The writers of client software should be aware that the software represents the user in their interactions over the Internet, and should be careful to allow the user to be aware of any actions they may take which may have an unexpected significance to themselves or others.

In particular, the convention has been established that the GET and HEAD methods should never have the significance of taking an action other than retrieval. These methods should be considered “safe.” This allows user agents to represent other methods, such as POST, PUT and DELETE, in a special way, so that the user is made aware of the fact that a possibly unsafe action is being requested.

Naturally, it is not possible to ensure that the server does not generate side-effects as a result of performing a GET request; in fact, some dynamic resources consider that a feature. The important distinction here is that the user did not request the side-effects, so therefore cannot be held accountable for them.

14.3 Abuse of Server Log Information

A server is in the position to save personal data about a user’s requests which may identify their reading patterns or subjects of interest. This information is clearly confidential in nature and its handling may be constrained by law in certain countries. People using the HTTP protocol to provide data are responsible for ensuring that such material is not distributed without the permission of any individuals that are identifiable by the published results.

14.4 Transfer of Sensitive Information

Like any generic data transfer protocol, HTTP cannot regulate the content of the data that is transferred, nor is there any a priori method of determining the sensitivity of any particular piece of information within the context of any given request. Therefore, applications should supply as much control over this information as possible to the provider of that information. Four header fields are worth special mention in this context: Server, Forwarded, Referer and From.

Revealing the specific software version of the server may allow the server machine to become more vulnerable to attacks against software that is known to contain security holes. Implementors should make the Server header field a configurable option.

Proxies which serve as a portal through a network firewall should take special precautions regarding the transfer of header information that identifies the hosts behind the firewall. In particular, they should remove, or replace with sanitized versions, any Forwarded fields generated behind the firewall.

The Referer field allows reading patterns to be studied and reverse links drawn. Although it can be very useful, its power can be abused if user details are not separated from the information contained in the Referer. Even when the personal information has been removed, the Referer field may indicate a private document’s URI whose publication would be inappropriate.

The information sent in the From field might conflict with the user’s privacy interests or their site’s security policy, and hence it should not be transmitted without the user being able to disable, enable, and modify the contents of the field. The user must be able to set the contents of this field within a user preference or application defaults configuration.

We suggest, though do not require, that a convenient toggle interface be provided for the user to enable or disable the sending of From and Referer information.

15. Acknowledgments

This specification makes heavy use of the augmented BNF and generic constructs defined by David H. Crocker for RFC 822 [9]. Similarly, it reuses many of the definitions provided by Nathaniel Borenstein and Ned Freed for MIME [7]. We hope that their inclusion in this specification will help reduce past confusion over the relationship between HTTP and Internet mail message formats.

The HTTP protocol has evolved considerably over the past four years. It has benefited from a large and active developer community—the many people who have participated on the *www-talk* mailing list—and it is that community which has been most responsible for the success of HTTP and of the World-Wide Web in general. Marc Andreessen, Robert Cailliau, Daniel W. Connolly, Bob Denny, John Franks, Jean-Francois Groff, Phillip M. Hallam-Baker, Håkon W. Lie, Ari Luotonen, Rob McCool, Lou Montulli, Dave Raggett, Tony Sanders, and Marc VanHeyningen deserve special recognition for their efforts in defining early aspects of the protocol.

This document has benefited greatly from the comments of all those participating in the HTTP-WG. In addition to those already mentioned, the following individuals have contributed to this specification:

```
Gary Adams                  Harald Tveit Alvestrand
Keith Ball                  Brian Behlendorf
Paul Burchard               Maurizio Codogno
Mike Cowlishaw              Roman Czyborra
Michael A. Dolan            Jim Gettys
Marc Hedlund                Koen Holtman
Alex Hopmann                Bob Jernigan
Shel Kaphan                 Rohit Khare
Martijn Koster              Alexei Kosut
Dave Kristol                Daniel LaLiberte
Paul Leach                  Albert Lunde
John C. Mallery             Jean-Philippe Martin-Flatin
Larry Masinter              Mitra
Jeffrey Mogul               Gavin Nicol
Bill Perry                  Jeffrey Perry
Owen Rees                   Luigi Rizzo
David Robinson              Marc Salomon
Rich Salz                   Jim Seidman
Chuck Shotton               Eric W. Sink
Simon E. Spero              Richard N. Taylor
Robert S. Thau              François Yergeau
Mary Ellen Zurko
```

16. References

[1] H. Alvestrand. “Tags for the identification of languages.” RFC 1766, UNINETT, March 1995.

[2] F. Anklesaria, M. McCahill, P. Lindner, D. Johnson, D. Torrey, B. Alberti. “The Internet Gopher Protocol: A distributed document search and retrieval protocol.” RFC 1436, University of Minnesota, March 1993.

[3] T. Berners-Lee. “Universal Resource Identifiers in WWW: A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World-Wide Web.” RFC 1630, CERN, June 1994.

[4] T. Berners-Lee, L. Masinter, M. McCahill. “Uniform Resource Locators (URL).” RFC 1738, CERN, Xerox PARC, University of Minnesota, December 1994.

[5] T. Berners-Lee, D. Connolly. “HyperText Markup Language Specification – 2.0.” RFC 1866, MIT/LCS, November 1995.

[6] T. Berners-Lee, R. Fielding, H. Frystyk. “Hypertext Transfer Protocol – HTTP/1.0.” Work in Progress (draft-ietf-http-v10-spec-04.txt), MIT/LCS, UC Irvine, September 1995.

[7] N. Borenstein, N. Freed. “MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies.” RFC 1521, Bellcore, Innosoft, September 1993.

[8] R. Braden. “Requirements for Internet hosts - application and support.” STD 3, RFC 1123, IETF, October 1989.

[9] D. H. Crocker. “Standard for the Format of ARPA Internet Text Messages.” STD 11, RFC 822, UDEL, August 1982.

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

[11] R. Fielding. “Relative Uniform Resource Locators.” RFC 1808, UC Irvine, June 1995.

[12] M. Horton, R. Adams. “Standard for interchange of USENET messages.” RFC 1036 (Obsoletes RFC 850), AT&T Bell Laboratories, Center for Seismic Studies, December 1987.

[13] B. Kantor, P. Lapsley. “Network News Transfer Protocol: A Proposed Standard for the Stream-Based Transmission of News.” RFC 977, UC San Diego, UC Berkeley, February 1986.

[14] K. Moore. “MIME (Multipurpose Internet Mail Extensions) Part Two: Message Header Extensions for Non-ASCII Text.” RFC 1522, University of Tennessee, September 1993.

[15] E. Nebel, L. Masinter. “Form-based File Upload in HTML.” RFC 1867, Xerox Corporation, November 1995.

[16] J. Postel. “Simple Mail Transfer Protocol.” STD 10, RFC 821, USC/ISI, August 1982.

[17] J. Postel. “Media Type Registration Procedure.” RFC 1590, USC/ISI, March 1994.

[18] J. Postel, J. K. Reynolds. “File Transfer Protocol (FTP).” STD 9, RFC 959, USC/ISI, October 1985.

[19] J. Reynolds, J. Postel. “Assigned Numbers.” STD 2, RFC 1700, USC/ISI, October 1994.

[20] K. Sollins, L. Masinter. “Functional Requirements for Uniform Resource Names.” RFC 1737, MIT/LCS, Xerox Corporation, December 1994.

[21] US-ASCII. Coded Character Set – 7-Bit American Standard Code for Information Interchange. Standard ANSI X3.4-1986, ANSI, 1986.

[22] ISO-8859. International Standard -- Information Processing --
8-bit Single-Byte Coded Graphic Character Sets --
Part 1: Latin alphabet No. 1, ISO 8859-1:1987.
Part 2: Latin alphabet No. 2, ISO 8859-2, 1987.
Part 3: Latin alphabet No. 3, ISO 8859-3, 1988.
Part 4: Latin alphabet No. 4, ISO 8859-4, 1988.
Part 5: Latin/Cyrillic alphabet, ISO 8859-5, 1988.
Part 6: Latin/Arabic alphabet, ISO 8859-6, 1987.
Part 7: Latin/Greek alphabet, ISO 8859-7, 1987.
Part 8: Latin/Hebrew alphabet, ISO 8859-8, 1988.
Part 9: Latin alphabet No. 5, ISO 8859-9, 1990.

17. Authors’ Addresses

Roy T. Fielding
Department of Information and Computer Science
University of California
Irvine, CA 92717-3425, U.S.A.
Fax: +1 (714) 824-4056
Email: fielding@ics.uci.edu

Henrik Frystyk Nielsen
W3 Consortium
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139, U.S.A.
Fax: +1 (617) 258 8682
Email: frystyk@w3.org

Tim Berners-Lee
Director, W3 Consortium
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139, U.S.A.
Fax: +1 (617) 258 8682
Email: timbl@w3.org

Appendices

These appendices are provided for informational reasons only — they do not form a part of the HTTP/1.1 specification.

A. Internet Media Type message/http

In addition to defining the HTTP/1.1 protocol, this document serves as the specification for the Internet media type "message/http". The following is to be registered with IANA [17].

```
Media Type name:          message
Media subtype name:       http
Required parameters:      none
Optional parameters:      version, msgtype
      version:  The HTTP-Version number of the enclosed message
                (e.g., "1.1"). If not present, the version can be
                determined from the first line of the body.
      msgtype:  The message type -- "request" or "response". If not
                present, the type can be determined from the first
                line of the body.
Encoding considerations: only "7bit", "8bit", or "binary" are
                          permitted
Security considerations: none
```

B. Tolerant Applications

Although this document specifies the requirements for the generation of HTTP/1.1 messages, not all applications will be correct in their implementation. We therefore recommend that operational applications be tolerant of deviations whenever those deviations can be interpreted unambiguously.

Clients should be tolerant in parsing the Status-Line and servers tolerant when parsing the Request-Line. In particular, they should accept any amount of SP or HT characters between fields, even though only a single SP is required.

The line terminator for HTTP-header fields is the sequence CRLF. However, we recommend that applications, when parsing such headers, recognize a single LF as a line terminator and ignore the leading CR.

C. Relationship to MIME

HTTP/1.1 reuses many of the constructs defined for Internet Mail (RFC 822 [9]) and the Multipurpose Internet Mail Extensions (MIME [7]) to allow entities to be transmitted in an open variety of representations and with extensible mechanisms. However, HTTP is not a MIME-compliant application. HTTP's performance requirements differ substantially from those of Internet mail. Since it is not limited by the restrictions of existing mail protocols and SMTP gateways, HTTP does not obey some of the constraints imposed by RFC 822 and MIME for mail transport.

This appendix describes specific areas where HTTP differs from MIME. Proxies/gateways to MIME-compliant protocols must be aware of these differences and provide the appropriate conversions where necessary.

C.1 Conversion to Canonical Form

MIME requires that an entity be converted to canonical form prior to being transferred, as described in Appendix G of RFC 1521 [7]. Although HTTP does require media types to be transferred in canonical form, it changes the definition of “canonical form” for text-based media types as described in Section 3.7.1.

C.1.1 Representation of Line Breaks

MIME requires that the canonical form of any text type represent line breaks as CRLF and forbids the use of CR or LF outside of line break sequences. Since HTTP allows CRLF, bare CR, and bare LF (or the octet sequence(s) to which they would be translated for the given character set) to indicate a line break within text content, recipients of an HTTP message cannot rely upon receiving MIME-canonical line breaks in text.

Where it is possible, a proxy/gateway from HTTP to a MIME-compliant protocol should translate all line breaks within text/* media types to the MIME canonical form of CRLF. However, this may be complicated by the presence of a Content-Encoding and by the fact that HTTP allows the use of some character sets which do not use octets 13 and 10 to represent CR and LF, as is the case for some multi-byte character sets. If canonicalization is performed, the Content-Length header field value must be updated to reflect the new body length.

C.1.2 Default Character Set

MIME requires that all subtypes of the top-level Content-Type "text" have a default character set of US-ASCII [21]. In contrast, HTTP defines the default character set for "text" to be ISO-8859-1 [22] (a superset of US-ASCII). Therefore, if a text/* media type given in the Content-Type header field does not already include an explicit charset parameter, the parameter

```
;charset="iso-8859-1"
```

should be added by the proxy/gateway if the entity contains any octets greater than 127.

C.2 Conversion of Date Formats

HTTP/1.1 uses a restricted subset of date formats to simplify the process of date comparison. Proxies/gateways from other protocols should ensure that any Date header field present in a message conforms to one of the HTTP/1.1 formats and rewrite the date if necessary.

C.3 Introduction of Content-Encoding

MIME does not include any concept equivalent to HTTP’s Content-Encoding header field. Since this acts as a modifier on the media type, proxies/gateways to MIME-compliant protocols must either change the value of the Content-Type header field or decode the Entity-Body before forwarding the message.

> ***Note:*** *Some experimental applications of Content-Type for Internet mail have used a media-type parameter of ";conversions=<content-coding>" to perform an equivalent function as Content-Encoding. However, this parameter is not part of the MIME specification at the time of this writing.*

C.4 No Content-Transfer-Encoding

HTTP does not use the Content-Transfer-Encoding (CTE) field of MIME. Proxies/gateways from MIME-compliant protocols must remove any non-identity CTE ("quoted-printable" or "base64") encoding prior to delivering the response message to an HTTP client. Proxies/gateways to MIME-compliant protocols are responsible for ensuring that the message is in the correct format and encoding for safe transport on that protocol, where "safe transport" is defined by the limitations of the protocol being used. At a minimum, the CTE field of

```
Content-Transfer-Encoding: binary
```

should be added by the proxy/gateway if it is unwilling to apply a content transfer encoding.

An HTTP client may include a Content-Transfer-Encoding as an extension Entity-Header in a POST request when it knows the destination of that request is a proxy/gateway to a MIME-compliant protocol.

C.5 Introduction of Transfer-Encoding

HTTP/1.1 introduces the Transfer-Encoding header field (Section 10.39). Proxies/gateways must remove any transfer coding prior to forwarding a message via a MIME-compliant protocol. The process for decoding the "chunked" transfer coding (Section 3.6) can be represented in pseudo-code as:

```
length := 0
read chunk-size and CRLF
while (chunk-size > 0) {
   read chunk-data and CRLF
   append chunk-data to Entity-Body
   length := length + chunk-size
   read chunk-size and CRLF
}
read entity-header
while (entity-header not empty) {
   append entity-header to existing header fields
   read entity-header
}
Content-Length := length
Remove "chunked" from Transfer-Encoding
```

D. Changes from HTTP/1.0

This section will summarize the differences between versions 1.0 and 1.1 of the Hypertext Transfer Protocol.