Network Working Group
Request for Comments: 1945
Category: Informational

T. Berners-Lee
MIT/LCS
R. Fielding
UC Irvine
H. Frystyk
MIT/LCS
May 1996

Hypertext Transfer Protocol -- HTTP/1.0

Status of This Memo

This memo provides information for the Internet community. This memo does not specify an Internet standard of any kind. Distribution of this memo is unlimited.

IESG Note:

The IESG has concerns about this protocol, and expects this document to be replaced relatively soon by a standards track document.

Abstract

The Hypertext Transfer Protocol (HTTP) is an application-level protocol with the lightness and speed necessary 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 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 reflects common usage of the protocol referred to as "HTTP/1.0".

Table of Contents

1. Introduction

1.1 Purpose

The Hypertext Transfer Protocol (HTTP) is an application-level protocol with the lightness and speed necessary for distributed, collaborative, hypermedia information systems. HTTP has been in use by the World-Wide Web global information initiative since 1990. This specification reflects common usage of the protocol referred too as "HTTP/1.0". This specification describes the features that seem to be consistently implemented in most HTTP/1.0 clients and servers. The specification is split into two sections. Those features of HTTP for which implementations are usually consistent are described in the main body of this document. Those features which have few or inconsistent implementations are listed in Appendix D.

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) [2], as a location (URL) [4] or name (URN) [16], 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 [7] and the Multipurpose Internet Mail Extensions (MIME) [5].

HTTP is also used as a generic protocol for communication between user agents and proxies/gateways to other Internet protocols, such as SMTP [12], NNTP [11], FTP [14], Gopher [1], and WAIS [8], allowing basic hypermedia access to resources available from diverse applications and simplifying the implementation of user agents.

1.2 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.

1.3 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. Some HTTP/1.0 applications use heuristics to describe what is or is not a "cachable" response, but these rules are not standardized.

On the Internet, HTTP communication generally takes place over TCP/IP connections. The default port is TCP 80 [15], 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/1.0 request and response structures onto the transport data units of the protocol in question is outside the scope of this specification.

Except for experimental applications, current practice requires that the connection be established by the client prior to each request and closed by the server after sending the response. Both clients and servers should be aware that either party may close the connection prematurely, due to user action, automated time-out, or program failure, and should handle such closing in a predictable fashion. In any case, the closing of the connection by either or both parties always terminates the current request, regardless of its status.

1.4 HTTP and MIME

HTTP/1.0 uses many of the constructs defined for MIME, as defined in RFC 1521 [5]. Appendix C describes the ways in which the context of HTTP allows for different use of Internet Media Types than is typically found in Internet mail, and gives the rationale for those differences.

2. Notational Conventions and Generic Grammar

2.1 Augmented BNF

All of the mechanisms specified in this document are described in both prose and an augmented Backus-Naur Form (BNF) similar to that used by RFC 822 [7]. 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 ("I") 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 [17].

```
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/1.0 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 3.6.

```
CRLF           = CR LF
```

HTTP/1.0 headers may be folded onto multiple lines if each 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 )
```

However, folding of header lines is not expected by some applications, and should not be generated by HTTP/1.0 applications.

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.

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

Recipients of header field TEXT containing octets outside the US-ASCII character set may assume that they represent ISO-8859-1 characters.

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/1.0 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 may 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>
```

Single-character quoting using the backslash ("\") character is not permitted in HTTP/1.0.

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.

```
    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.

This document defines both the 0.9 and 1.0 versions of the HTTP protocol. Applications sending Full-Request or Full-Response messages, as defined by this specification, must include an HTTP-Version of "HTTP/1.0".

HTTP/1.0 servers must:

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

HTTP/1.0 clients must:

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

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 or respond with an error. 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 [2], and finally the combination of Uniform Resource Locators (URL) [4] and Names (URN) [16]. 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 [9], 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 [9]. 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 (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

HTTP/1.0 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 [6] (an update to RFC 822 [7]). The second format is in common use, but is based on the obsolete RFC 850 [10] date format and lacks a four-digit year. HTTP/1.0 clients and servers that parse the date value should accept all three formats, though they must never generate the third (asctime) format.

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/1.0 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.

3.4 Character Sets

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

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

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

HTTP character sets are identified by case-insensitive tokens. The complete set of tokens are defined by the IANA Character Set registry [15]. 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 [5] -- the US-ASCII [17] and ISO-8859 [18] 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 [15] must represent the character set defined

by that registry. Applications should limit their use of character sets to those defined by the IANA registry.

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

3.5 Content Codings

Content coding values are used to indicate an encoding transformation that has been 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 = "x-gzip" | "x-compress" | token
```

> Note: For future compatibility, HTTP/1.0 applications should consider "gzip" and "compress" to be equivalent to "x-gzip" and "x-compress", respectively.

All content-coding values are case-insensitive. HTTP/1.0 uses content-coding values in the Content-Encoding (Section 10.3) header field. 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:

x-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.

x-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 Media Types

HTTP uses Internet Media Types [13] in the Content-Type header field (Section 10.5) in order to provide open and extensible data typing.

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

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

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

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

Some older HTTP applications do not recognize media type parameters. HTTP/1.0 applications should only use media type parameters when they are necessary to define the content of a message.

Media-type values are registered with the Internet Assigned Number Authority (IANA [15]). The media type registration process is outlined in RFC 1590 [13]. Use of non-registered media types is discouraged.

3.6.1 Canonicalization and Text Defaults

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

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

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

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

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

3.6.2 Multipart Types

MIME provides for a number of "multipart" types -- encapsulations of several entities within a single message's Entity-Body. The multipart types registered by IANA [15] do not have any special meaning for HTTP/1.0, though user agents may need to understand each type in order to correctly interpret the purpose of each body-part. An HTTP user agent should follow the same or similar behavior as a MIME user agent does upon receipt of a multipart type. HTTP servers should not assume that all HTTP clients are prepared to handle multipart types.

All multipart types share a common syntax 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. Multipart body-parts may contain HTTP header fields which are significant to the meaning of that part.

3.7 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).

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/1.0 messages
               | Full-Response
```

Full-Request and Full-Response use the generic message format of RFC 822 [7] 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 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 [7]. Each header field consists of a name followed immediately by a colon (":"), a single space (SP) character, and the field value. Field names are case-insensitive. Header fields can be extended over multiple lines by preceding each extra line with at least one SP or HT, though this is not recommended.

```
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 = Date                     ; Section 10.6
               | Pragma                   ; Section 10.12
```

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/1.0 server receives a Simple-Request, it must respond with an HTTP/0.9 Simple-Response. An HTTP/1.0 client capable of receiving a Full-Response should 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         = "GET"                    ; Section 8.1
               | "HEAD"                   ; Section 8.2
               | "POST"                   ; Section 8.3
               | extension-method

extension-method = token
```

The list of methods acceptable by a specific resource can change dynamically; the client is notified through the return code of the response if a method is not allowed on a resource. Servers should return the status code 501 (not implemented) if the method is unrecognized or not implemented.

The methods commonly used by HTTP/1.0 applications are fully defined 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 two options for Request-URI are dependent on the nature of the request.

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 Expires header field. 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/1.0
```

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/1.0
```

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).

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 = Authorization            ; Section 10.2
               | From                     ; Section 10.8
               | If-Modified-Since        ; Section 10.9
               | Referer                  ; Section 10.13
               | User-Agent               ; Section 10.15
```

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/1.0 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/1.0 200 "), 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:

- o 1xx: Informational - Not used, but reserved for future use
- o 2xx: Success - The action was successfully received, understood, and accepted.
- o 3xx: Redirection - Further action must be taken in order to complete the request
- o 4xx: Client Error - The request contains bad syntax or cannot be fulfilled
- o 5xx: Server Error - The server failed to fulfill an apparently valid request

The individual values of the numeric status codes defined for HTTP/1.0, 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    = "200"   ; OK
                   | "201"   ; Created
                   | "202"   ; Accepted
                   | "204"   ; No Content
                   | "301"   ; Moved Permanently
                   | "302"   ; Moved Temporarily
                   | "304"   ; Not Modified
                   | "400"   ; Bad Request
                   | "401"   ; Unauthorized
                   | "403"   ; Forbidden
                   | "404"   ; Not Found
                   | "500"   ; Internal Server Error
                   | "501"   ; Not Implemented
                   | "502"   ; Bad Gateway
                   | "503"   ; Service Unavailable
                   | extension-code

    extension-code = 3DIGIT

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

HTTP status codes are extensible, but the above codes are the only ones generally recognized in current practice. HTTP applications are not required to understand the meaning of all registered status

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

6.2 Response Header Fields

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

```
    Response-Header = Location                ; Section 10.11
                    | Server                  ; Section 10.14
                    | WWW-Authenticate        ; Section 10.16
```

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 10.1
                   | Content-Encoding         ; Section 10.3
                   | Content-Length           ; Section 10.4
                   | Content-Type             ; Section 10.5
                   | Expires                  ; Section 10.7
                   | Last-Modified            ; Section 10.10
                   | 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 header field in the request message headers. HTTP/1.0 requests containing an entity body must include a valid Content-Length header field.

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


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

A Content-Type specifies the media type of the underlying data. A Content-Encoding may be used to indicate any additional content coding applied to the type, usually for the purpose of data compression, that is a property of the resource requested. The default for the content encoding is none (i.e., the identity function).

Any HTTP/1.0 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 two 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 closing of the connection by the server.

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. Therefore, HTTP/1.0 requests containing an entity body must include a valid Content-Length header field. If a request contains an entity body and Content-Length is not specified, and the server does not recognize or cannot calculate the length from other fields, then the server should send a 400 (bad request) response.

> Note: Some older servers supply an invalid Content-Length when sending a document that contains server-side includes dynamically inserted into the data stream. It must be emphasized that this will not be tolerated by future versions of HTTP. Unless the client knows that it is receiving a response from a compliant server, it should not depend on the Content-Length value being correct.

8. Method Definitions

The set of common methods for HTTP/1.0 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.

8.1 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 changes 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 10.9. The conditional GET method is intended to reduce network usage by allowing cached entities to be refreshed without requiring multiple requests or transferring unnecessary data.

8.2 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.

There is no "conditional HEAD" request analogous to the conditional GET. If an If-Modified-Since header field is included with a HEAD request, it should be ignored.

8.3 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 [3], 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.

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.

A valid Content-Length is required on all HTTP/1.0 POST requests. An HTTP/1.0 server should respond with a 400 (bad request) message if it cannot determine the length of the request message's content.

Applications must not cache responses to a POST request because the application has no way of knowing that the server would return an equivalent response on some future request.

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. HTTP/1.0 does not define any 1xx status codes and they are not a valid response to a HTTP/1.0 request. However, they may be useful for experimental applications which are outside the scope of this specification.

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.

201 Created

The request has been fulfilled and resulted in a new resource being created. The newly created resource can be referenced by the URI(s) returned in the entity of the response. 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).

Of the methods defined by this specification, only POST can create a resource.

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.

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 scripts or other 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.

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 subsequent 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

This response code is not directly used by HTTP/1.0 applications, but serves as the default for interpreting the 3xx class of responses.

The requested resource is available at one or more locations. 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. If the server has a preferred choice, it should include the URL in a Location field; user agents may use this field value for automatic redirection.

301 Moved Permanently

The requested resource has been assigned a new permanent URL and any future references to this resource should be done using that URL. Clients with link editing capabilities should automatically relink references to the Request-URI to the new reference returned by the server, where possible.

The new URL must be given by the Location field in the response. Unless it was a HEAD request, the Entity-Body of the response should contain a short note with a hyperlink to the new URL.

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

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

302 Moved Temporarily

The requested resource resides temporarily under a different URL. Since the redirection may be altered on occasion, the client should continue to use the Request-URI for future requests.

The URL must be given by the Location field in the response. Unless it was a HEAD request, the Entity-Body of the response should contain a short note with a hyperlink to the new URI(s).

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

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

304 Not Modified

If the client has performed a conditional GET request and access is allowed, but the document has not been modified since the date and time specified in the If-Modified-Since field, the server must respond with this status code and not send an Entity-Body to the client. Header fields contained in the response should only include information which is relevant to cache managers or which may have changed independently of the entity's Last-Modified date. Examples of relevant header fields include: Date, Server, and Expires. A cache should update its cached entity to reflect any new field values given in the 304 response.

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 10.16) containing a challenge applicable to the requested resource. The client may repeat the request with a suitable Authorization header field (Section 10.2). 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.

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.

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.

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

10. Header Field Definitions

This section defines the syntax and semantics of all commonly used HTTP/1.0 header fields. For general and entity header fields, both sender and recipient refer to either the client or the server, depending on who sends and who receives the message.

RFC 1945 HTTP/1.0 May 1996

10.1 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. 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
```

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.

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 by the server.

10.2 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.

10.3 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 coding has been applied to the resource, and thus what decoding mechanism must be applied in order to obtain the media-type referenced by the Content-Type header field. The 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" ":" content-coding
```

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

```
Content-Encoding: x-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.

10.4 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/1.0 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 a response 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.

10.5 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 3.6. An example of the field is

```
Content-Type: text/html
```

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

10.6 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 POST request, 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.

10.7 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/1.0, a robust implementation is always desirable.

10.8 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 [7] (as updated by RFC 1123 [6]):

```
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.

10.9 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 shall 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.

10.10 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.

10.11 Location

The Location response-header field defines the exact location of the resource that was identified by the Request-URI. For 3xx responses, the location must indicate the server's preferred URL for automatic redirection to the resource. Only one absolute URL is allowed.

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

An example is

```
Location: http://www.w3.org/hypertext/WWW/NewLocation.html
```

10.12 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 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.

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.

10.13 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.

10.14 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 3.7) 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.

> 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.

RFC 1945 HTTP/1.0 May 1996

> Note: Some existing servers fail to restrict themselves to the product token syntax within the Server field.

10.15 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 3.7) 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
```

> Note: Some current proxy applications append their product information to the list in the User-Agent field. This is not recommended, since it makes machine interpretation of these fields ambiguous.

> Note: Some existing clients fail to restrict themselves to the product token syntax within the User-Agent field.

10.16 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 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 403 (forbidden) response.

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/1.0 does not provide a means for a client to be authenticated with a proxy.

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 authorize 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 [5] encoded string in the credentials.

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

basic-cookie      = <base64 [5] 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.

12. Security Considerations

This section is meant to inform application developers, information providers, and users of the security limitations in HTTP/1.0 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.

12.1 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/1.0 does not prevent additional authentication schemes and encryption mechanisms from being employed to increase security.

12.2 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, 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.

12.3 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.

12.4 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. Three header fields are worth special mention in this context: Server, 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.

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.

12.5 Attacks Based On File and Path Names

Implementations of HTTP origin servers should be careful to restrict the documents returned by HTTP requests to be only those that were intended by the server administrators. If an HTTP server translates HTTP URIs directly into file system calls, the server must take special care not to serve files that were not intended to be delivered to HTTP clients. For example, Unix, Microsoft Windows, and other operating systems use ".." as a path component to indicate a directory level above the current one. On such a system, an HTTP server must disallow any such construct in the Request-URI if it would otherwise allow access to a resource outside those intended to be accessible via the HTTP server. Similarly, files intended for reference only internally to the server (such as access control files, configuration files, and script code) must be protected from inappropriate retrieval, since they might contain sensitive information. Experience has shown that minor bugs in such HTTP server implementations have turned into security risks.

13. Acknowledgments

This specification makes heavy use of the augmented BNF and generic constructs defined by David H. Crocker for RFC 822 [7]. Similarly, it reuses many of the definitions provided by Nathaniel Borenstein and Ned Freed for MIME [5]. We hope that their inclusion in this specification will help reduce past confusion over the relationship between HTTP/1.0 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, Jean-Francois Groff, Phillip M. Hallam-Baker, Hakon W. Lie, Ari Luotonen, Rob McCool, Lou Montulli, Dave Raggett, Tony Sanders, and Marc VanHeyningen deserve special recognition for their efforts in defining aspects of the protocol for early versions of this specification.

Paul Hoffman contributed sections regarding the informational status of this document and Appendices C and D.

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	John Franks
Jim Gettys	Marc Hedlund
Koen Holtman	Alex Hopmann
Bob Jernigan	Shel Kaphan
Martijn Koster	Dave Kristol
Daniel LaLiberte	Paul Leach
Albert Lunde	John C. Mallery
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	Robert S. Thau
Francois Yergeau	Mary Ellen Zurko
Jean-Philippe Martin-Flatin	

14. References

[1] Anklesaria, F., McCahill, M., Lindner, P., Johnson, D., Torrey, D., and B. Alberti, "The Internet Gopher Protocol: A Distributed Document Search and Retrieval Protocol", RFC 1436, University of Minnesota, March 1993.

[2] Berners-Lee, T., "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.

[3] Berners-Lee, T., and D. Connolly, "Hypertext Markup Language - 2.0", RFC 1866, MIT/W3C, November 1995.

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

[5] Borenstein, N., and 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.

[6] Braden, R., "Requirements for Internet hosts - Application and Support", STD 3, RFC 1123, IETF, October 1989.

[7] Crocker, D., "Standard for the Format of ARPA Internet Text Messages", STD 11, RFC 822, UDEL, August 1982.

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

[9] Fielding, R., "Relative Uniform Resource Locators", RFC 1808, UC Irvine, June 1995.

[10] Horton, M., and R. Adams, "Standard for interchange of USENET Messages", RFC 1036 (Obsoletes RFC 850), AT&T Bell Laboratories, Center for Seismic Studies, December 1987.

[11] Kantor, B., and 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.

[12] Postel, J., "Simple Mail Transfer Protocol." STD 10, RFC 821, USC/ISI, August 1982.

[13] Postel, J., "Media Type Registration Procedure." RFC 1590, USC/ISI, March 1994.

[14] Postel, J., and J. Reynolds, "File Transfer Protocol (FTP)", STD 9, RFC 959, USC/ISI, October 1985.

[15] Reynolds, J., and J. Postel, "Assigned Numbers", STD 2, RFC 1700, USC/ISI, October 1994.

[16] Sollins, K., and L. Masinter, "Functional Requirements for Uniform Resource Names", RFC 1737, MIT/LCS, Xerox Corporation, December 1994.

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

[18] 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.

15. Authors' Addresses

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

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

Appendices

These appendices are provided for informational reasons only -- they do not form a part of the HTTP/1.0 specification.

A. Internet Media Type message/http

In addition to defining the HTTP/1.0 protocol, this document serves as the specification for the Internet media type "message/http". The following is to be registered with IANA [13].

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., "1.0"). 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/1.0 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/1.0 uses many of the constructs defined for Internet Mail (RFC 822 [7]) and the Multipurpose Internet Mail Extensions (MIME [5]) to allow entities to be transmitted in an open variety of representations and with extensible mechanisms. However, RFC 1521 discusses mail, and HTTP has a few features that are different than those described in RFC 1521. These differences were carefully chosen to optimize performance over binary connections, to allow greater freedom in the use of new media types, to make date comparisons easier, and to acknowledge the practice of some early HTTP servers and clients.

At the time of this writing, it is expected that RFC 1521 will be revised. The revisions may include some of the practices found in HTTP/1.0 but not in RFC 1521.

This appendix describes specific areas where HTTP differs from RFC 1521. Proxies and gateways to strict MIME environments should be aware of these differences and provide the appropriate conversions where necessary. Proxies and gateways from MIME environments to HTTP also need to be aware of the differences because some conversions may be required.

C.1 Conversion to Canonical Form

RFC 1521 requires that an Internet mail entity be converted to canonical form prior to being transferred, as described in Appendix G of RFC 1521 [5]. Section 3.6.1 of this document describes the forms allowed for subtypes of the "text" media type when transmitted over HTTP.

RFC 1521 requires that content with a Content-Type of "text" represent line breaks as CRLF and forbids the use of CR or LF outside of line break sequences. HTTP allows CRLF, bare CR, and bare LF to indicate a line break within text content when a message is transmitted over HTTP.

Where it is possible, a proxy or gateway from HTTP to a strict RFC 1521 environment should translate all line breaks within the text media types described in Section 3.6.1 of this document to the RFC 1521 canonical form of CRLF. Note, however, that this may be complicated by the presence of a Content-Encoding and by the fact that HTTP allows the use of some character sets which do not use octets 13 and 10 to represent CR and LF, as is the case for some multi-byte character sets.

C.2 Conversion of Date Formats

HTTP/1.0 uses a restricted set of date formats (Section 3.3) to simplify the process of date comparison. Proxies and gateways from other protocols should ensure that any Date header field present in a message conforms to one of the HTTP/1.0 formats and rewrite the date if necessary.

C.3 Introduction of Content-Encoding

RFC 1521 does not include any concept equivalent to HTTP/1.0's Content-Encoding header field. Since this acts as a modifier on the media type, proxies and gateways from HTTP to MIME-compliant protocols must either change the value of the Content-Type header field or decode the Entity-Body before forwarding the message. (Some experimental applications of Content-Type for Internet mail have used a media-type parameter of ";conversions=<content-coding>" to perform an equivalent function as Content-Encoding. However, this parameter is not part of RFC 1521.)

C.4 No Content-Transfer-Encoding

HTTP does not use the Content-Transfer-Encoding (CTE) field of RFC 1521. Proxies and gateways from MIME-compliant protocols to HTTP must remove any non-identity CTE ("quoted-printable" or "base64") encoding prior to delivering the response message to an HTTP client.

Proxies and gateways from HTTP to MIME-compliant protocols are responsible for ensuring that the message is in the correct format and encoding for safe transport on that protocol, where "safe transport" is defined by the limitations of the protocol being used. Such a proxy or gateway should label the data with an appropriate Content-Transfer-Encoding if doing so will improve the likelihood of safe transport over the destination protocol.

C.5 HTTP Header Fields in Multipart Body-Parts

In RFC 1521, most header fields in multipart body-parts are generally ignored unless the field name begins with "Content-". In HTTP/1.0, multipart body-parts may contain any HTTP header fields which are significant to the meaning of that part.

D. Additional Features

This appendix documents protocol elements used by some existing HTTP implementations, but not consistently and correctly across most HTTP/1.0 applications. Implementors should be aware of these features, but cannot rely upon their presence in, or interoperability

with, other HTTP/1.0 applications.

D.1 Additional Request Methods

D.1.1 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.

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 data to be processed. 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 should not apply the request to some other resource.

D.1.2 DELETE

The DELETE method requests that the origin server delete the resource identified by the Request-URI.

D.1.3 LINK

The LINK method establishes one or more Link relationships between the existing resource identified by the Request-URI and other existing resources.

D.1.4 UNLINK

The UNLINK method removes one or more Link relationships from the existing resource identified by the Request-URI.

D.2 Additional Header Field Definitions

D.2.1 Accept

The Accept request-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.

D.2.2 Accept-Charset

The Accept-Charset request-header field can be used to indicate a list of preferred character sets other than the default US-ASCII and ISO-8859-1. 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.

D.2.3 Accept-Encoding

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

D.2.4 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.

D.2.5 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.

D.2.6 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.

D.2.7 MIME-Version

HTTP 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, as defined by RFC 1521 [5], should indicate that the message is MIME-conformant. Unfortunately, some older HTTP/1.0 servers send it indiscriminately, and thus this field should be ignored.

D.2.8 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.

D.2.9 Title

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

D.2.10 URI

The URI entity-header field may contain some or all of the Uniform Resource Identifiers (Section 3.2) by which the Request-URI resource can be identified. There is no guarantee that the resource can be accessed using the URI(s) specified.