HTTP Working Group
INTERNET-DRAFT
<draft-ietf-http-v10-spec-02.ps>
Expires February 13, 1996

T. Berners-Lee, MIT/W3C
R. Fielding, UC Irvine
H. Frystyk, MIT/W3C
August 13, 1995

Hypertext Transfer Protocol — HTTP/1.0

Status of this Memo

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

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

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

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

Abstract

The Hypertext Transfer Protocol (HTTP) is an application-level protocol 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 and negotiation of data representation, allowing systems to be built independently of the data being transferred.

HTTP has been in use by the World-Wide Web global information initiative since 1990. This specification reflects common usage of the protocol referred to as "HTTP/1.0".

Table of Contents

1. Introduction....4
   1.1 Purpose....4
   1.2 Overall Operation....4
   1.3 Terminology....5
2. Notational Conventions and Generic Grammar....6
   2.1 Augmented BNF....6
   2.2 Basic Rules....7
3. Protocol Parameters....9
   3.1 HTTP Version....9
   3.2 Uniform Resource Identifiers....10
      3.2.1 General Syntax....10
      3.2.2 http URL....11
   3.3 Date/Time Formats....11
   3.4 Coded Character Sets....12
   3.5 Encoding Mechanisms....13
   3.6 Media Types....13
      3.6.1 Canonicalization and Text Defaults....14
      3.6.2 Multipart Types....15
   3.7 Product Tokens....15
4. HTTP Message....16
   4.1 Message Types....16
   4.2 Message Headers....16
   4.3 General Message Header Fields....17
5. Request....18
   5.1 Request-Line....18
   5.2 Method....18
      5.2.1 GET....19
      5.2.2 HEAD....19
      5.2.3 POST....19
   5.3 Request-URI....20
   5.4 Request Header Fields....20
6. Response....21
   6.1 Status-Line....21
   6.2 Status Codes and Reason Phrases....21
      6.2.1 Informational 1xx....22
      6.2.2 Successful 2xx....23
      6.2.3 Redirection 3xx....24
      6.2.4 Client Error 4xx....25
      6.2.5 Server Errors 5xx....26
   6.3 Response Header Fields....26
7. Entity....27
   7.1 Entity Header Fields....27
   7.2 Entity Body....27
      7.2.1 Type....27
      7.2.2 Length....28

8. Header Field Definitions 29
8.1 Allow 29
8.2 Authorization 29
8.3 Content-Encoding 29
8.4 Content-Length 30
8.5 Content-Type 30
8.6 Date 30
8.7 Expires 31
8.8 From 32
8.9 If-Modified-Since 32
8.10 Last-Modified 33
8.11 Location 33
8.12 MIME-Version 33
8.13 Pragma 34
8.14 Referer 34
8.15 Server 35
8.16 User-Agent 35
8.17 WWW-Authenticate 35
9. Access Authentication 36
9.1 Basic Authentication Scheme 37
10. Security Considerations 38
10.1 Authentication of Clients 38
10.2 Idempotent Methods 38
10.3 Abuse of Server Log Information 38
10.4 Transfer of Sensitive Information 38
11. Acknowledgments 40
12. References 41
13. Authors' Addresses 42
Appendix A. Internet Media Type message/http 43
Appendix B. Tolerant Applications 43
Appendix C. Relationship to MIME 44
C.1 Conversion to Canonical Form 44
C.1.1 Representation of Line Breaks 44
C.1.2 Default Coded Character Set 44
C.2 Conversion of Date Formats 44
C.3 Introduction of Content-Encoding 45
C.4 No Content-Transfer-Encoding 45

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 to as "HTTP/1.0". This specification is not intended to become an Internet standard; rather, it defines those features of the HTTP protocol that can reasonably be expected of any implementation which claims to be using HTTP/1.0.

Practical information systems require more functionality than simple retrieval, including search, front-end update, and annotation. HTTP/1.0 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/1.0 is also used for communication between user agents and various gateways, allowing hypermedia access to existing Internet protocols like SMTP [12], NNTP [11], FTP [14], Gopher [1], and WAIS [8]. HTTP/1.0 is designed to allow such gateways, via proxy servers, without any loss of the data conveyed by those earlier protocols.

1.2 Overall Operation

The HTTP protocol is based on a request/response paradigm. A requesting program (termed a client) establishes a connection with a receiving program (termed 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 its protocol version and a success or error code, followed by a MIME-like message containing server information, entity metainformation, and possible body content. It should be noted that a given program may be capable of being both a client and a server; our use of those terms refers only to the role being performed by the program during a particular connection, rather than to the program's purpose in general.

On the Internet, the communication generally takes place over a TCP/IP connection. The default port is TCP 80 [15], but other ports can be used. This does not preclude the HTTP/1.0 protocol 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.

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 must be capable of handling cases where either party closes the connection prematurely, due to user action, automated time-out, or program failure. In any case, the closing of the connection by either or both parties always terminates the current request, regardless of its status.

1.3 Terminology

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

connection
: A virtual circuit established between two parties for the purpose of communication.

message
: A structured sequence of octets transmitted via the connection as the basic component of communication.

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 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
: A program that establishes connections for the purpose of sending requests.

user agent
: The client program which is closest to the user and which initiates requests at their behest. These are often browsers, editors, spiders (web-traversing robots), or other end user tools.

server
: A 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 forwarding requests. Proxies are often used to act as a portal through a network firewall. A *proxy server* accepts requests from other clients and services them either internally or by passing them, with possible translation, on to other servers. A *caching proxy* is a proxy server with a local cache of server responses — some requested resources can be serviced from the cache rather than from the origin server. Some proxy servers also act as origin servers.

gateway
: A proxy which services HTTP requests by translation into protocols other than HTTP. The reply sent from the remote server to the gateway is likewise translated into HTTP before being forwarded to the user agent.

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 ("|") 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, zero or more 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. 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 the continuation lines begin with linear whitespace characters. 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.

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.

```
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.

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 coded 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 coded character set may assume that they represent ISO-8859-1 characters.

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:

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

HTTP/1.0 clients must:

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

Proxies must be careful in forwarding requests that are received in a format different than that of the proxy's native version. Since the protocol version indicates the protocol capability of the sender, a proxy 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 must either downgrade the request version or respond with an error. Requests with a version lower than that of the proxy's native format may be upgraded by the proxy before being forwarded; the proxy's response to that request must follow the normal server requirements.

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/1.0 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
hex           = "A" | "B" | "C" | "D" | "E" | "F"
              |  "a" | "b" | "c" | "d" | "e" | "f" | DIGIT

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

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        = <FQDN or IP address, as defined in RFC 1738>
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.

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 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/1.0 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 Coded 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.

However, this is more commonly referred to as a coded character set. Coded 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 coded character set, we define here the preferred names for those coded character sets most likely to be used with HTTP entities. This set of charset values includes those registered by RFC 1521 [5] — the US-ASCII [17] and ISO-8859 [18] coded 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 coded character set defined by that registry. Applications are encouraged, but not required, to limit their use of coded character sets to those defined by the IANA registry.

3.5 Encoding Mechanisms

Encoding mechanism values are used to indicate an encoding transformation that has been or can be applied to a resource. Encoding mechanisms 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.

```
encoding-mechanism          = "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 encoding-mechanism values are case-insensitive. HTTP/1.0 uses encoding-mechanism values in the Content-Encoding (Section 8.3) header field. Although the value describes the encoding-mechanism, 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 encoding-mechanism 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. Gzip is available from the GNU project at <URL:ftp://prep.ai.mit.edu/pub/gnu/>.

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] (formerly referred to as MIME Content-Types [5]) in order to provide open and extensible data typing and type negotiation. For mail applications, where there is no type negotiation between sender and receiver, it is reasonable to put strict limits on the set of allowed media types. With HTTP, where the sender and recipient can communicate directly, applications are allowed more freedom in the use of non-registered types. The following grammar for media types is a superset of that for MIME because it does not restrict itself to the official IANA and x-token types.

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

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

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

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

Many current applications do not recognize media type parameters. Since parameters are a fundamental aspect of media types, this must be considered an error in those applications. Nevertheless, HTTP/1.0 applications should only use media type parameters when they are necessary to define the content of a message.

If a given media-type value has been registered by the IANA, any use of that value must be indicative of the registered data format. Although HTTP allows the use of non-registered media types, such usage must not conflict with the IANA registry. Data providers are strongly encouraged to register their media types with IANA via the procedures outlined in RFC 1590 [13].

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

3.6.1 Canonicalization and Text Defaults

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

HTTP redefines the canonical form of text media to allow multiple octet sequences to indicate a text line break. In addition to the preferred form of CRLF, HTTP applications must accept a bare CR or LF alone as representing a single line break in text media. Furthermore, if the text media is represented in a coded character set which does not use octets 13 and 10 for CR and LF respectively, as is the case for some multi-byte coded character sets, HTTP allows the use of whatever octet sequence(s) is defined by that coded character set to represent the equivalent of CRLF, bare CR, and bare LF. It is assumed that any recipient capable of using such a coded character set will know the appropriate octet sequence for representing line breaks within that coded character set.

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

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

HTTP also redefines the default coded character set for text media in an entity body. If a textual media type defines a charset parameter with a registered default value of "US-ASCII", HTTP changes the default to be "ISO-8859-1". Since the ISO-8859-1 [18] coded character set is a superset of US-ASCII [17], this has no effect upon the interpretation of entity bodies which only contain octets within the US-ASCII set (0 - 127). The presence of a charset parameter value in a Content-Type header field overrides the default.

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

3.6.2 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. Ideally, an HTTP user agent should follow the same or similar behavior as a MIME user agent does upon receipt of a multipart type.

As in MIME [5], 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. Unlike in MIME, 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 (a.k.a. "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 5.4
                  | 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 6.3
                  | 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 5.4), Response-Header (Section 6.3), 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 LWS, though this is not recommended.

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

field-name     = 1*<any CHAR, excluding CTLs, SP, and ":">
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 Message 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 communicating parties or the content being transferred. These headers apply only to the message being transmitted.

```
General-Header  = Date                  ; Section 8.6
                | MIME-Version          ; Section 8.12
                | Pragma                ; Section 8.13
```

General header field names can be extended only via a change in the protocol version. Unknown 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 requested, 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 5.4
                 |  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.

5.2 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" | "HEAD" | "POST"
               |  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 unknown or not implemented.

The set of common methods for HTTP/1.0 is described below. Although this set can be easily expanded, additional methods cannot be assumed to share the same semantics for separately extended clients and servers.

5.2.1 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 8.9. 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.

5.2.2 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.

5.2.3 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.

5.3 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 server. The proxy is requested to forward the request and return the response. If the request is idempotent and a response is cached, the proxy may return 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 origin 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/hypertext/WWW/TheProject.html HTTP/1.0
```

The most common form of Request-URI is that used to identify a resource on an origin server. 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 /hypertext/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).

5.4 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. All header fields are optional and conform to the generic HTTP-header syntax.

```
Request-Header  = Authorization             ; Section 8.2
                | From                      ; Section 8.8
                | If-Modified-Since         ; Section 8.9
                | Referer                   ; Section 8.14
                | User-Agent                ; Section 8.16
```

Request-Header field names can be extended only via a change in the protocol version. Unknown 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 6.3
                   | 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.

6.2 Status Codes and Reason Phrases

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

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

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

The individual values of the numeric status codes defined for HTTP/1.0, 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.

```
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 unknown response as being equivalent to the x00 status code of that class. For example, if an unknown status code of 421 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 are encouraged to present the entity returned with the response to the user, since that entity is likely to include human-readable information which will explain the unusual status.

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

6.2.1 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.

6.2.2 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 being 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 is encouraged, but not obliged, to actually create the resource before using this Status-Code. If the action cannot be carried out immediately, or within a clearly defined timeframe, the server should respond with 202 (accepted) instead.

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.

6.2.3 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 can sometimes be carried out by the user agent without interaction with the user, but it is strongly recommended that this only take place if the method used in the request is idempotent (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 the Location 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 are encouraged to 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. The Entity-Body of the response should contain a short hypertext 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.

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. The Entity-Body of the response should contain a short hypertext note with a hyperlink to the new URI(s).

If the 302 status code is received in response to a request 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.

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 shall 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 and which may have changed independently of the entity's Last-Modified date. Examples of relevant header fields include: Date, Server, and Expires.

6.2.4 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 is encouraged to 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 is discouraged from repeating the request without modifications.

401 Unauthorized

The request requires user authentication. The response must include a WWW-Authenticate header field (Section 8.17) containing a challenge applicable to the requested resource. The client may repeat the request with a suitable Authorization header field. HTTP access authentication is explained in Section 9.

403 Forbidden

The server understood the request, but is refusing to perform the request for an unspecified reason. Authorization will not help and the request should not be repeated. This status code can be used if the server does not want to make public why the request has not been fulfilled.

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.

6.2.5 Server Errors 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 is encouraged to 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 received an invalid response from the gateway or 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.*

6.3 Response Header Fields

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

```
Response-Header= Location                ; Section 8.11
               | Server                  ; Section 8.15
               | WWW-Authenticate        ; Section 8.17
```

Response-Header field names can be extended only via a change in the protocol version. Unknown 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 8.1
                  | Content-Encoding         ; Section 8.3
                  | Content-Length           ; Section 8.4
                  | Content-Type             ; Section 8.5
                  | Expires                  ; Section 8.7
                  | Last-Modified            ; Section 8.10
                  | extension-header

extension-header = HTTP-header
```

The extension-header mechanism allows additional Entity-Header to be defined without changing the protocol, but these fields cannot be assumed to be recognizable by the recipient. Unknown 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/1.0 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. This specification defines one request method, POST, that allows an entity-body. In general, 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 content 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 content header fields may lead one to believe they do. The responses 204 (no content) and 304 (not modified) must not include a message body.

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 encoding mechanism 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).

The Content-Type header field has no default value. If and **only if** the media type is not given by a Content-Type header, as is always the case for Simple-Response messages, the receiver may attempt to guess the media type via inspection of its content and/or the name extension(s) of the URL used to specify the resource. If the media type remains unknown, the receiver should treat it as type "`application/octet-stream`".

7.2.2 Length

When an Entity-Body is included with a message, the length of that body may be determined in one of several ways. If a Content-Length header field is present, its value in bytes represents the length of the Entity-Body. Otherwise, the body length is determined by the 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 content 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. Header Field Definitions

This section defines the syntax and semantics of all standard HTTP/1.0 header fields. For Entity-Header fields, both *sender* and *recipient* refer to either the client or the server, depending on who sends and who receives the entity.

8.1 Allow

The Allow 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 field value should be followed. This field has no default value; if left undefined, the set of allowed methods is defined by the origin server at the time of each request.

A proxy must not modify the allow header 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.

8.2 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 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 9. If a request is authenticated and a realm specified, the same credentials should be valid for all other requests within this realm.

Proxies must not cache the response to a request containing an Authorization field.

8.3 Content-Encoding

The Content-Encoding header field is used as a modifier to the media-type. When present, its value indicates what additional encoding mechanism 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" ":" encoding-mechanism
```

Encoding mechanisms 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.

8.4 Content-Length

The Content-Length 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
```

Although it is not required, applications are strongly encouraged to use this field to indicate the size of the Entity-Body to be transferred, regardless of the media type of the entity.

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

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

8.5 Content-Type

The Content-Type 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
```

The Content-Type header field has no default value. Further discussion of methods for identifying the media type of an entity is provided in Section 7.2.1.

8.6 Date

The Date header 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 default 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 receiver if and only if the message will be cached by that receiver or gatewayed via a protocol which requires a Date.

Only one Date header field is allowed per message. 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.*

8.7 Expires

The Expires field gives the date/time after which the entity should be considered stale. This allows information providers to suggest the volatility of the resource. Caching clients, including proxies, must not cache this copy of the resource beyond the date given, unless its status has been updated by a later check of the origin server. 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 are strongly encouraged to 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
```

The Expires field has no default value. 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, copies of 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. 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.

> ***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.*

8.8 From

The From 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.*

8.9 If-Modified-Since

The If-Modified-Since 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.

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 the 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.

8.10 Last-Modified

The Last-Modified 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 receiver should interpret it: if the receiver 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.

8.11 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
```

8.12 MIME-Version

HTTP is not a MIME-conformant protocol (see Appendix C). However, HTTP/1.0 messages may include a single MIME-Version header field to indicate what version of the MIME protocol was used to construct the message. Use of the MIME-Version header field should indicate that the message is in full compliance with the MIME protocol (as defined in [5]). Unfortunately, current versions of HTTP/1.0 clients and servers use this field indiscriminately, and thus receivers must not take it for granted that the message is indeed in full compliance with MIME. Gateways are responsible for ensuring this compliance (where possible) when exporting HTTP messages to strict MIME environments. Future HTTP/1.0 applications must only use MIME-Version when the message is intended to be MIME-conformant.

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

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

8.13 Pragma

The Pragma message header field is used to specify directives that should be applied to all intermediaries along the request/response chain. The directives typically specify behavior intended to prevent intermediate proxies or caches from adversely interfering with the request or response. All pragma directives specify optional behavior from the viewpoint of the protocol; however, some systems may require that behavior be consistent with the directives. HTTP/1.0 defines semantics for the "`no-cache`" directive.

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

When the "`no-cache`" directive is present in a request message, a caching intermediary must 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, regardless of their significance to that proxy, since the directives may be applicable to all intermediaries along the request/response chain. It is not possible to specify a pragma for a specific proxy; however, any pragma directive not relevant to a proxy should be ignored.

Pragma directives do not apply to the end-points of a request/response chain. For example, a user agent's internal (non-shared) cache and/or history mechanism should ignore all pragma directives in received messages. Similarly, pragma directives are not applicable to the origin of a resource, though they may be applicable to a server's internal response cache.

8.14 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.*

8.15 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 should 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.*

8.16 User-Agent

The User-Agent 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 always include this field with requests. The field can contain multiple product tokens (Section 3.7) 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
```

The User-Agent field may include additional information within comments.

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.*

8.17 WWW-Authenticate

The WWW-Authenticate 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 9. 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.

9. 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 at the transport level, via message encapsulation, and/or 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.

9.1 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 are encouraged to implement the scheme in order to communicate with servers that use it.

10. 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.

10.1 Authentication of Clients

As mentioned in Section 9.1, 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.

10.2 Idempotent 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" and should not have side effects. This allows the client software to represent other methods, such as POST, in a special way so that the user is made aware of the fact that an non-idempotent 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.

10.3 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.

10.4 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 are encouraged to 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 are encouraged to 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.

11. 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 three 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, Håkon W. Lie, Ari Luotonen, Rob McCool, 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.

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	Albert Lunde
John C. Mallery	Larry Masinter
Mitra	Gavin Nicol
Bill Perry	Jeffrey Perry
Owen Rees	David Robinson
Marc Salomon	Rich Salz
Jim Seidman	Chuck Shotton
Eric W. Sink	Simon E. Spero
Robert S. Thau	François Yergeau
Mary Ellen Zurko	

12. References

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

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

[3] T. Berners-Lee and D. Connolly. "HyperText Markup Language Specification – 2.0." Work in Progress (draft-ietf-html-spec-05.txt), MIT/W3C, August 1995.

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

[5] N. Borenstein 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] R. Braden. "Requirements for Internet hosts - application and support." STD 3, RFC 1123, IETF, October 1989.

[7] D. H. Crocker. "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] R. Fielding. "Relative Uniform Resource Locators." RFC 1808, UC Irvine, June 1995.

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

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

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

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

[16] K. Sollins 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.

13. Authors' Addresses

Tim Berners-Lee
Director, W3 Consortium
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139, U.S.A.
Tel: +1 (617) 253 5702
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.
Tel: +1 (714) 824-4049
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.
Tel: +1 (617) 258 8143
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 StatusLine and servers tolerant when parsing the RequestLine. 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 reuses 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, HTTP is not a MIME-conforming application. HTTP's performance requirements differ substantially from those of Internet mail. Since it is not limited by the restrictions of existing mail protocols and gateways, HTTP does not obey some of the constraints imposed by RFC 822 and MIME for mail transport.

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

C.1 Conversion to Canonical Form

MIME requires that an entity be converted to canonical form prior to being transferred, as described in Appendix G of RFC 1521 [5]. Although HTTP does require media types to be transferred in canonical form, it changes the definition of "canonical form" for text-based media types as described in Section 3.6.1.

C.1.1 Representation of Line Breaks

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

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

C.1.2 Default Coded Character Set

MIME requires that all subtypes of the top-level Content-Type "text" have a default coded character set of US-ASCII [17]. In contrast, HTTP defines the default coded character set for "text" to be ISO-8859-1 [18] (a superset of US-ASCII). Therefore, if a text/* media type given in the Content-Type header field does not already include an explicit charset parameter, the parameter

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

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

C.2 Conversion of Date Formats

HTTP/1.0 uses a restricted subset of date formats to simplify the process of date comparison. 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

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

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

C.4 No Content-Transfer-Encoding

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

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

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

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